目录

第一章 产品简介

华恒 HHARM740 套件是一套基于 WINBODN W90N740 处理器的嵌入式 LINUX 开发平台。华恒 HHARM740 硬件平台由核心板（HHARM740 Rev1.0）和底板（外设板或称基本板）组成，核心板上集成 W90N740 处理器，16M SDRAM 以及 2M 的 FLASH，为用户的软件研发提供了足够的空间。

Winbond740 是一款采用 ARM7 内核的 16/32 位精简指令嵌入式微处理器。芯片内部集成了 2 个 10/100M 以太网控制器 MAC、SDRAM 控制器、1 个串行接口控制器、PWM 控制器、I2C 控制器、IIS 控制器、实时时钟、AD 转换等丰富的外围控制模块，这些接口模块位系统板集成各种功能提供了可能，对初窥门径的嵌入式系统的爱好者和学习者来说，W90N740 无疑是非常合适的一款产品；另外，如同本平台所实现的一样，通过外界网络控制芯片，也可以实现各种网络通讯协议。

核心板和底板配合即构成一个最小的完整应用系统。系统具有体积小、耗电低、处理能力强等特点，能够装载和运行嵌入式 Linux 操作系统。用户可以在这个系统平台上进行自主软件开发，并对 HHARM740 进行测试和评估，也可在保持核心板不变的情况下，针对具体的应用通过对底板的更改来实现定制自己的应用系统。HHARM740 套件中提供底板硬件电路图及硬件设计文档，并在本手册中给出所有管脚说明及信号定义，极大的方便了用户进行硬件扩展开发。

华恒 HHARM740 套件提供完备的嵌入式 LINUX 开发环境及丰富的开发调测工具软件。

W90N740 提供了许多嵌入式应用产品所共有的外围产品，如 SDRAM 控制器、以太网 MAC、DMA、计时器、UART、芯片选择、通用 I/O 以及片上存储，所有这些都采取高效节省的方式，从而减少系统成本，加速系

统设计。其主要资源为：

2.5V 核电压
外部存储器控制器
以太网 MAC 器
2 路 DMA
2 路 UART
1 路 I2C
1 路 I2S
6 路定时器
看门狗
71 路 I/O 口
8 路中断
电源管理模块
8 路 10 位 ADC
实时时钟
片上 PLL

处理器产品规格：

* Up to 80MHz
* 2.5-V 核电源，3V 端口电源
* 160 LQFP

华恒 HHARM740 套件之硬件主要构造：

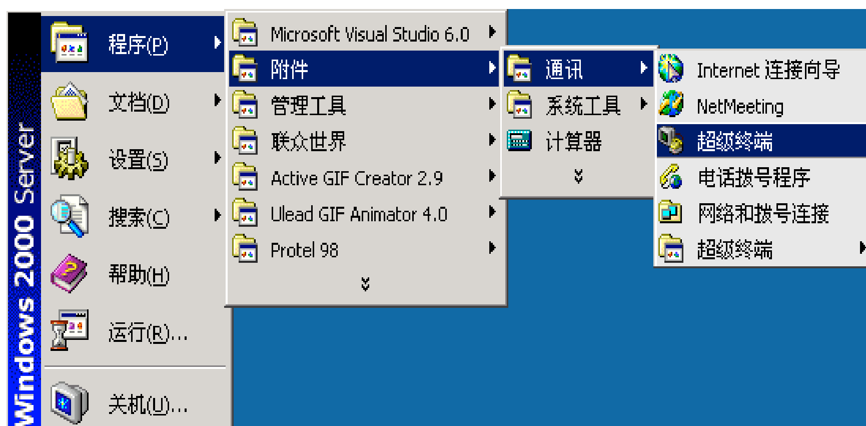
- W90N740 处理器
- 4Mbytes 16 位 FLASH
- 16Mbytes 16 位 SDRAM
- RS-232 标准串口（2 个）
- JTAG 接口
- 9V 直流电源

- H/W 复位建
- 电源状态指示
- 100M 以太网

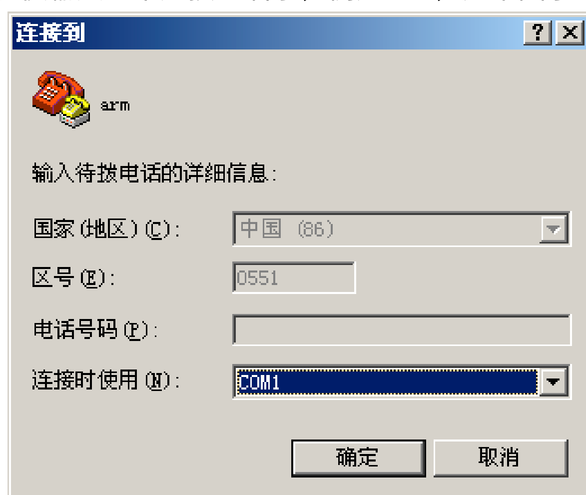
套件的标准配置由核心板、底板（外设接口板）、9V 直流电源变压器、JTAG 接口卡及 JTAG 线、标准 PC 串口线、软件安装光盘、技术手册以及配置清单组成。

套件简单检测：

用户拿到套件后，打开包装，用串口线将开发板的串口与一台 WINDOWS PC 机的串口 1 连接起来（不要插 JTAG），在“开始”—>“程序”—>“附件”—>“通讯”中打开超级终端，



随便输入一个连接的名字，例如 arm，如下图示：



选择 COM1，波特率 115200，无硬件流控。



然后将 9V 直流电源接到板子上，这时在超级终端便可看到板子上 bios 控制界面的信息打印出来。稍作等待便会出现嵌入的 Linux 操作系统的启动信息，并最终进入提示符状态. 这是检测套件软硬件工作是否正常的最为简捷的方法。

若以上检测有问题，请立即与华恒公司联系：0551－5325631 或 0551－5325173，华恒公司将立即给您更换产品。

【注意】

串口线、JTAG 线都**严禁带电拔插！**

第二章 软件系统

HHARM740 为一台采用 WINBONDW90N740X01 处理器、安装有 LINUX 操作系统的开发原型机，其功用相当于一台装有 REDHAT LINUX，装有串口的 PC 机。对于 HHARM740,它提供的所有软件（操作系统和应用软件）都固化在板上两片 FLASH 内，就相当于 PC 机的硬盘。FLASH 上的内容可通过烧写工具来更新升级。用户可为 HHARM740 开发应用程序或更改其上的操作系统工作方式（因为操作系统是开放源代码的），和在 PC 上开发应用唯一的不同之处在于它要采用一种交叉编译的开发模式，即为 HHARM740 开发应用，不能直接在 HHARM740 板子上编辑、编译和调试，而必须把这些工作寄宿到另一台 PC 机上去完成。详细介绍请参见后面章节。

随着微处理器的产生，价格低廉、结构小巧的 CPU 和外设连接提供了稳定可靠的硬件架构，那么限制嵌入式系统发展的瓶颈就突出表现在了软件方面。尽管从八十年代末开始，陆续出现了一些嵌入式操作系统，比较著名的有 Vxwork、pSOS、Neculeus 和 Windows CE。但这些专用操作系统都是商业化产品，其高昂的价格使许多低端产品的小公司望而却步；而且，源代码封闭性也大大限制了开发者的积极性。另外，结合国内实情，当前国家对自主操作系统的大力支持，也为源码开放的 LINUX 的推广提供的广阔的发展前景。还有，对上层应用开发者而言，嵌入式系统需要的是一套高度简练、界面友善、质量可靠、应用广泛、易开发、多任务，并且价格低廉的操作系统。在不久的将来，从冰箱到收音机都会内置处理器。因为 Linux 的开放性，许多人认为 Linux 非常适合多数 Internet 设备。他们认为 Linux 可以支持不同的设备，支持不同的配置。Linux 对厂商不偏不倚而且成本极低，能够很快成为用于各种设备的操作系统。如今，业界已经达成共识：即嵌入式 linux 是大势所趋，其巨大的市场潜力与酝酿的无限商机必然会吸引众多的厂商进入这一领域。

嵌入式操作系统主要有 Palm OS，Windows CE，EPOC，LinuxCE，QNX，ECOS，LYNX，高端嵌入式系统要求许多高级的功能，如图形用户界面和网络支持 。很多高端 RTOS 供应商已经提供了这些功能，但其价格也很高端，一般人难以接受。微软的 Windows CE 也有此类功能，却不具备大多数嵌入式系统要求的实时性能，而且难以移植，也曾经有人想以 DOS 为基础用单独的第三方工具拼凑一个系统，但这种努力将是白费。现在需要的是一个便宜、成熟并且提供高端嵌入式系统所必须特性的操作系统，嵌入式 Linux 操作系统以价格低廉、功能强大又易于移植而正在被广泛采用，成为新兴的力量，所以，众多商家纷纷转向了嵌入式 LINUX。

Linux 为嵌入操作系统提供了一个极有吸引力的选择，它是个和 Unix 相似、以核心为基础的、完全内存保护、多任务多进程的操作系统。支持广泛的计算机硬件,包括 MOTOROLA，X86，Alpha，Sparc，MIPS，PPC，ARM，NEC 等现有的大部分芯片。软件源码全部公开，任何人可以修改并在 GNU 通用公共许可证(GNU General Public License)下发行,这样，开发人员可以对操作系统进行定制，再也不必担心像 MS WINDOWS 操作系统中“后门”的威胁。同时由于有 GPL 的控制,大家开发的东西大都相互兼容，不会走向分裂之路。Linux 用户遇到问题时可以通过 Internet 向网上成千上万的 Linux 开发者请教,这使最困难的问题也有办法解决。

Linux 带有 Unix 用户熟悉的完善的开发工具，几乎所有的 Unix 系统的应用软件都已移植到了 Linux 上。Linux 还提供了强大的网络功能，有多种可选择窗口管理器（X windows）。其强大的语言编译器 gcc、g++等也可以很容易得到。不但成熟完善、而且使用方便。

嵌入式系统选择 linux 的原因 ：

可应用于多种硬件平台。Linux 已经被移植到多种硬件平台，这对受开销、时间限制的研究与开发项目是很有吸引力的。原型可以在标准平台上开发然后移植到具体的硬件上，加快了软件与硬件的开发过程。

Linux 可以随意地配置不需要任何的许可证或商家的合作关系。唯一的限制是开发者必须做出对 Linux 社区有益的改动。

它是免费的，源代码可以得到。这是最吸引人的。毫无疑问，这会节省大量的开发费用。

微内核直接提供网络支持，而不必象其他操作系统要外挂 TCP/IP 协议包。

Linux 的高度模块化使添加部件非常容易。

Linux 在台式机上的成功，也保证了 Linux 在嵌入式系统中的辉煌前景。

HHARM740 开发板上提供的 LINUX 操作系统为一种专为嵌入式 NOMMU 微处理器定制的操作系统：uClinux。

Linux 是一种很受欢迎的操作系统，它与 UNIX 系统兼容，开放源代码。它原本被设计为桌面系统，现在广泛应用于服务器领域。而更大的影响在于它正逐渐的应用于嵌入式系统领域。uClinux 正是在这种氛围下产生的。在 uClinux 这个英文单词中 u 表示 Micro，小的意思，C 表示 Control，控制的意思，所以 uClinux 就是 Micro-Control-Linux，字面上的理解就是“针对微控制领域而设计的 Linux 系统”。

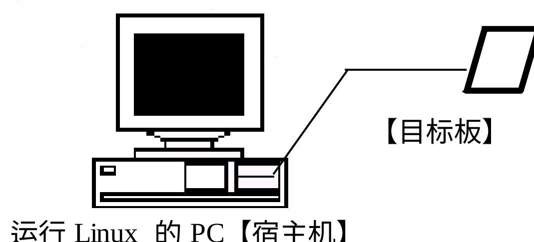
LINUX 是一个自由开放的世界，在 LINUX（无论 PC 还是嵌入式系统）上进行软件开发都可以在广袤的网络资源中获取帮助。下面是 uClinux 开发常用的网络资源站点：

www.uclinux.org

2.1 使用说明

2.1.1 建立宿主机开发环境

绝大多数的 Linux 软件开发都是以 native 方式进行的，即本机（HOST）开发、调试，本机运行的方式。这种方式通常不适合于嵌入式系统的软件开发，因为对于嵌入式系统的开发，没有足够的资源在本机（即板子上系统）运行开发工具和调试工具。通常的嵌入式系统的软件开发采用一种交叉编译调试的方式。交叉编译调试环境建立在宿主机（即一台 PC 机）上，对应的开发板叫做目标板。



开发时使用宿主机上的交叉编译、汇编及连接工具形成可执行的二进制代码，（这种可执行代码并不能在宿主机上执行，而只能在目标板上执行。）然后把可执行文件下载到目标机上运行。一般调试的方法包括串口调试和以太网口调试。对于 HHARM740，目前可采用串口调试。宿主机和目标板的处理器一般都不相同，宿主机为 INTEL 处理器，而目标板如

HHARM740 为 WINBOND W90N740X01，GNU 编译器提供这样的功能，在编译编译器时可以选择开发所需的宿主机和目标机从而建立开发环境。所以在进行嵌入式开发前第一步的工作就是要安装一台装有指定操作系统的 PC 机作宿主开发机，对于嵌入式 LINUX，宿主机上的操作系统一般要求为 REDHAT LINUX。然后要在宿主机上建立交叉编译调试的开发环境。环境的建立需要许多的软件模块协同工作，这将是一个比较繁杂的工作，但现在已完全由套件中光盘的安装而自动完成了。

下面逐步介绍常用的安装步骤：

在一台 PC 上安装 LINUX，建议 RedHat 系列，例如 REDHAT9.0 等。

建议选择 Custom 定制安装，在选择软件 Package 时选择最后一项：everything，即完全安装。

将我们附带的光盘插入 CDROM，然后执行以下命令：

mount /dev/cdrom /mnt

若系统不识别/dev/cdrom 的话，可以用如下命令，假设 CDROM 为从盘，即为/dev/hdb，则：

mount –t iso9660 /dev/hdb /mnt

cd /mnt ：进入 mount 后的目录

如果您的 CDROM 已经在安装 RedHat 的时候已经默认安装，以上命令请不要执行，请直接进入 CDROM 所在目录。

开始安装 HHARM740 软件

./ucinst ：执行安装程序

敲入 y，回车。

【注意】要进入中文环境只是为了能够看到安装启动时的一些中文提示信息，若没有中文环境也无所谓，只是看到一些乱码而已，用户只需按下 y 回车即可完成全部安装。

执行完毕后，会在根目录下生成工作目录：**/HHARM740**，内含 LINUX 内核、应用程序源代码以及各个工具软件。

嵌入式开发通常要求宿主机配置有网络，支持 NFS（为交叉开发时

mount 所用），支持 TFTP 服务器（为下载烧写所用）等等。然后要在宿主机上建立交叉编译调试的开发环境。环境的建立需要许多的软件模块协同工作，这将是一个比较繁杂的工作，但现在已完全由套件中光盘的安装而自动完成了。

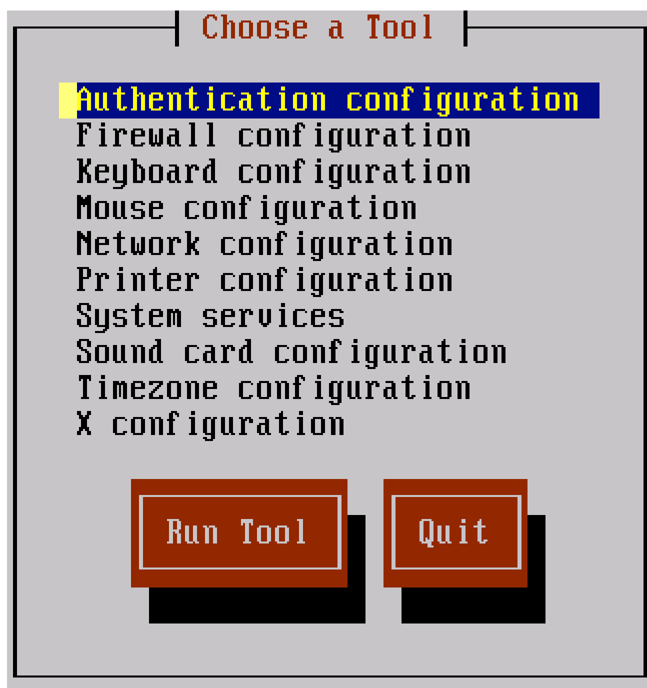
下面逐步介绍常用的安装步骤：

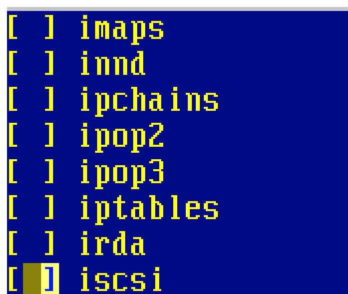
在一台 PC 上安装 LINUX，建议 RedHat 系列。安装时必须完全安装，即选择 Custom 定制安装，在配置 Firewall 时就选择 No Firewall，在选择软件 Package 时选择最后一项：everything，即完全安装。

配置好网络，TFTP 服务和 NFS（Enable Running）。

网络配置主要是要安装好以太网卡，对于一般常见的 RTL8139 网卡，REDHAT7.2 及更高版本可以自动识别并自动安装好，完全不要用户参与，因此建议使用该网卡。然后配置宿主机 IP：

ifconfig eth0 192.168.2.113





TFTP 服务的配置：

1、检查宿主机端的 TFTP 服务是否已经开通（此步骤只在第一次时使用，以后可忽略）：

开通宿主机上的 TFTP 服务，对于 REDHAT6.x，可以在宿主机上：

vim /etc/inetd.conf

查找 tftp，若发现前面有#就表示这一行被注释掉了，即服务没有打开，去掉#就打开了 TFTP 服务，然后重启宿主机即可。

对于 REDHAT7.2 以上的版本（例如经典的 REDHAT9.0），则在宿主机上执行 setup，选择 System services，将其中的 tftp 一项选中（出现 [*] 表示选中），并去掉 ipchains 和 iptables 两项服务（即去掉它们前面的*号）。然后还要选择 Firewall configuration，选中 No firewall。最后，退出 setup，执行如下命令以启动 TFTP 服务：

service xinetd restart

配置完成后，建议简单测试一下 TFTP 服务器是否可用，即自己 tftp 自己，例如在宿主机上执行：

cp /HHARM2410-R3/Images/zImage /tftpboot/

tftp 192.168.2.199

tftp>get zImage

若出现如下信息：

Received 741512 bytes in 0.7 seconds

就表示 TFTP 服务器配置成功了。若弹出信息说：Timed out，则表明未成功，或者用如下命令查看 tftp 服务是否开通：

netstat -a|grep tftp

若 TFTP 服务器没有配置成功，需要按照上述步骤重新检查一遍。

配置 NFS：

运行 linuxconf，

【注意】

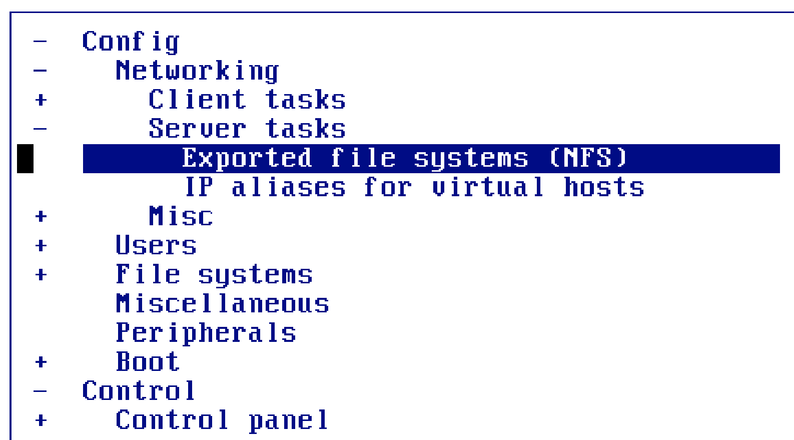
对于 REDHAT7.3 以后的版本，就没有 linuxconf 这个命令了，要么从华恒的产品光盘上安装它的 RPM（linuxconf-1.25r7-3.i386.rpm）包来提供 linuxconf 命令，要么就用后面介绍的一个更为简单的方法直接开

通 NFS 服务。

下面先介绍有 linuxconf 命令的方式：

在 config 选项下选 Server tasks，选中

Exported File systems(NFS)，见下图：



然后选择 Add Directory，加入根目录/，然后 Accept。

```
One exported file system

Path to export     /
Comment (opt)
Client name(s)
                   [ ] May write
                   [ ] Root privileges
                   [X] Request access from secure port
Client name(s)
                   [ ] May write
                   [ ] Root privileges
                   [X] Request access from secure port

  Accept   Cancel   Add   Del   Help
```

输出根目录/允许 NFS mount 后，配置界面显示如下：

```
Exported file systems
Setup file systems available for client hosts
Those systems will be accessible through NFS

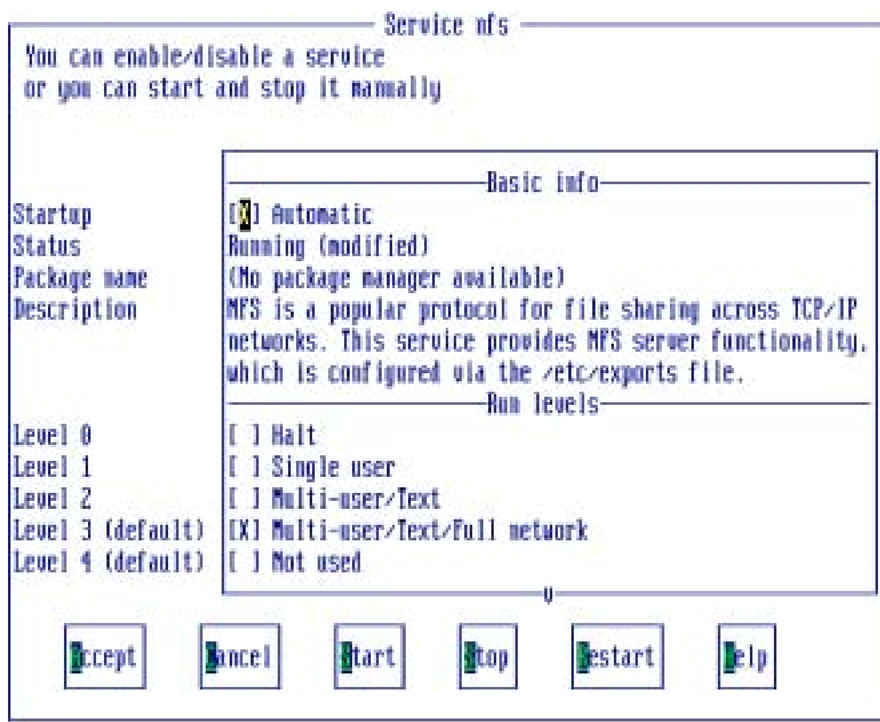
Select [Add] to add a new definition

        Directory Host More
        /

   Dismiss     Add     Help
```

选择 Dismiss（或者按下 ESC）退出这个界面。

下一步再选择 Control 项下面 Control panel 下的 Control Service activity，（界面见下图），然后选择 nfs enabled，然后 start。



配置好后的界面显示如下，其中 nfs 必须为： Automatic Running。

```
lpd             Manual
mars-nwe        Manual
mcserv          Manual
mysqld          Manual
named           Manual
netfs           Automatic Running
network         Automatic Running
nfs             Automatic Running (modified)
nfslock         Automatic Running
nscd            Manual
```

【注意】

这里建议把 ipchains 和 iptables 都取消其自动运行的状态：见下图：

```
identd          Manual
innd            Manual
ipchains        Manual
iptables        Manual
irda            Manual
iscsi           Manual
isdn            Manual
junkbuster      Manual
kadmin          Manual
kdcrotate       Manual
```

最后，在 Control 项下面 Control panel 下选择 Activate configuration，则弹出如下界面，提示系统配置的改动，选择 " Do it ",

```
    Changing permissions  of file /tmp from 40755 to 41777
The following command told me something had to be done
/etc/rc3.d/S10network probe
    Executing: /etc/rc3.d/S10network reload
Service smb is not running
    Executing: /etc/rc3.d/S91smb start

        Do nothing          Do it          Help
```

最后退出时则完成 NFS 配置。

```
+  Config
-  Control
-    Control panel
       Activate configuration
       Shutdown/Reboot
       Control service activity
+      Mount/Unmount file systems
       Configure superuser scheduled tasks
       Archive configurations
       Switch system profile
+    Linuxconf management
     Date & time
+  Status

      Quit        Act/Changes        Help
```

配置好后，界面应显示如下：

```
mars-nwe     Manual
mcserv       Manual
mysqld       Manual
named        Manual
netfs        Automatic Running
network      Automatic Running
nfs          Automatic Running (modified)
nfslock      Automatic Running
nscd         Manual
ntpd         Manual
```

至此配置完成。

下面再介绍一种更为直接简单的方法：

首先在 REDHAT LINUX PC 机上执行 setup，弹出菜单界面后，选中：System services，回车进入系统服务选项菜单，在其中选中 [*]nfs ，然后退出 setup 界面返回到命令提示符下。

vim　/etc/exports

将这个默认的空文件修改为只有如下一行内容：

/ (rw)

然后保存退出（:wq），然后执行如下命令：

/etc/rc.d/init.d/nfs restart

Shutting down NFS mountd:　　[　**OK**　]

Shutting down NFS daemon:

[　**OK**　]

Shutting down NFS quotas:

[　**OK**　]

Shutting down NFS services:

[　**OK**　]

Starting NFS services:

[**OK**]

Starting NFS quotas:

[**OK**]

Starting NFS daemon:

[**OK**]

Starting NFS mountd:

[**OK**]

这样就一切 OK 了！

配置完成后，可用如下办法简单测试一下 NFS 是否配置好了：

在宿主机上自己 mount 自己，看是否成功就可以判断 NFS 是否配好了。例如在宿主机/目录下执行：

mount 192.168.2.32:/ /mnt

然后到/mnt/目录下看是否可以列出/目录下的所有文件和目录，可以则说明 mount 成功，NFS 配置成功。

2.1.2 文件与目录结构

软件光盘安装后宿主机下目录如下：

[root@localhost HHARM740]# ls

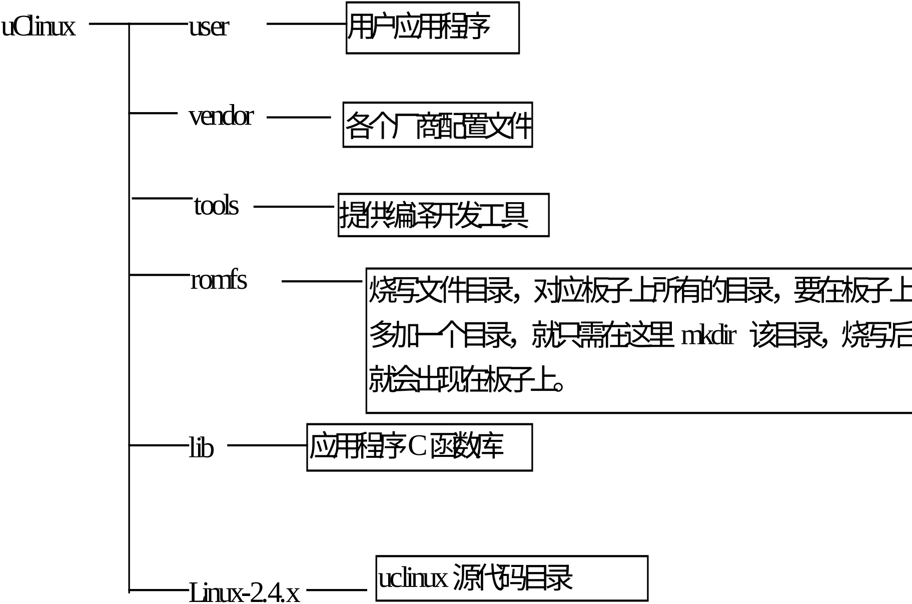
bios ***jtag-rw*** ***image*** minirc.dfl ***tools*** ***uClinux ucinst***

目录内容如下：

bios/	bootloader 映像文件目录。
jtag-rw/	JTAG 烧写工具目录。
Image/	板子出厂所烧制 uClinux 内核映像文件目录。
tools/	编译器文件目录

uClinux/　　uClinux 内核及应用程序目录。
minirc.dfl　　minicom 配置文件
ucinst　　自动安装文件

其中 uClinux 目录内容结构如下图：



其中 user 目录为各个应用程序的父目录，其下各个应用对应的目录结构如下：

目录	软件包描述
user/init/	init 进程
user/sash/	SHELL

目录	软件包描述
user/gui/	LCD 中英文显示例程 textout
user/demo/	板子启动时文字位图显示/滚屏演示例程
user/lissa/	lissa 曲线
user/sea/	位图显示
user/microwin/	microwindow
user/memtools/	内存（SDRAM/FLASH2）读写工具
user/boa/	嵌入式 LINUX 自带 WEB SERVER
user/route/	route/ifconfig 等网络命令
user/dhcpcd/	DHCP client
user/diald/	智能拨号 PROXY
user/chat	脚本拨号程序
user/pppd/	PPP 拨号
user/gdbserver/	gdb 远程调试服务器
user/busybox/	NFS mount/umount 等 SHELL 命令集
user/ttytest，serialtest	串口通信样例代码
user/ping/	ping 工具
user/key	INT1 所接按键的测试代码
user/tinykb	4×4 键盘测试软件，可显示到 LED 上。
user/adc	A/D 测试程序
user/handpad	触摸屏测试程序

上述诸多应用程序的源代码都在光盘中提供，但并未包含在标准发行版本中。只有其中少数几个工具加入编译并烧到板子上（可在 uClinux/romfs/bin/下面看到所有加入编译的应用程序），若用户需要，可把在 user 下增删定制自己所需的应用程序。具体参见后面 2.2.4 节介绍。

板子启动后的目录结构列表见下图：

bin dev etc home lib mnt proc tmp usr var

它完全就是 PC 机上 uClinux/romfs 目录下的内容。

因为板子上烧制的是一个 LINUX 操作系统，所以它的文件目录结构

和宿主机 PC LINUX 基本类似，目录的意义作用也基本类似。但板上文件系统和 PC 系统区别还是非常大的，例如板上文件系统只有几百 K 字节，而 PC LINUX 文件系统则有几个 G 之大。下面列表对比一下：

目录	开发板	PC LINUX
bin	含应用程序的可执行文件	
dev	含设备驱动所使用的设备文件	
etc	含启动脚本及各个应用程序的配置文件	
home	无用	多用户的工具目录
lib	无用	系统库
mnt	用于加载其它文件系统	用于加载其它文件系统
proc	proc 文件系统	
tmp	RAM 盘	临时文件目录
usr	无用	重要的系统目录，含用户安装的应用程序及 LINUX 内核源代码
var	RAM 盘	临时文件目录

其中/bin/目录用于存放可执行程序。/dev/是所有存放所有设备文件的目录。例如用户添加自己的设备驱动时，要为该设备创建设备文件，例如增加 JFFS 文件系统支持时，为其作设备驱动，要用到/dev/flash0～flash3 等设备名，首先要在宿主机 uClinux/romfs/dev/目录下用 touch（适用于 uClinux2.4 内核）创建，命令如下：

touch @flash0 b 60 16

这样在该目录下就会看到这个设备文件：

brwxr-xr-x 1 root root 60, 16 Feb 3 12:18 @flash0

gdbtftpflash/flash 烧写板子后，在板子的/dev/目录下也可以看到：

brw------- 1 0 0 60, 16 Jan 01 1970 flash0

/etc/目录下重要的文件就是系统启动 rc 脚本文件，其内容如下（TODO）：

[root@localhost etc]# cat rc

hostname HHARM740

/bin/expand /etc/ramfs.img /dev/ram0
mount -t proc proc /proc
mount -t ext2 /dev/ram0 /var
mkdir /var/tmp
mkdir /var/log
mkdir /var/run
mkdir /var/lock
cat /etc/motd

【系统一启动就执行用户应用程序即可类似的放在 rc 脚本最后】

2.1.3 交叉编译

宿主机上交叉编译开发环境由 GNU 开发工具集构成。它们在光盘安装时自动复制到 PC 机/usr/loca/bin/目录下。

GNU 工具集			
arm-elf-addr2line	arm-elf-gdb	arm-elf-run	elf2flt
arm-elf-ar	arm-elf-ld	arm-elf-size	cksum
arm-elf-as	arm-elf-nm	arm-elf-strings	genromfs
arm-elf-c++filt	arm-elf-objcopy	arm-elf-strip	
arm-elf-gasp	arm-elf-objdump		
arm-elf-gcc	arm-elf-ranlib		

嵌入式 LINUX 内核及设备驱动全部源代码（光盘安装后建立完备的开发环境）均位于 uClinux 目录下，整个软件系统编译在 uClinux 下执行：

```
make dep
make clean
make
```

若要更改内核配置或者应用软件的添加/删除，建议使用：

```
make menuconfig
```

来进行配置选择。

一 内核配置



进入 uClinux 目录，执行 make menuconfig，则出现如下主选单：

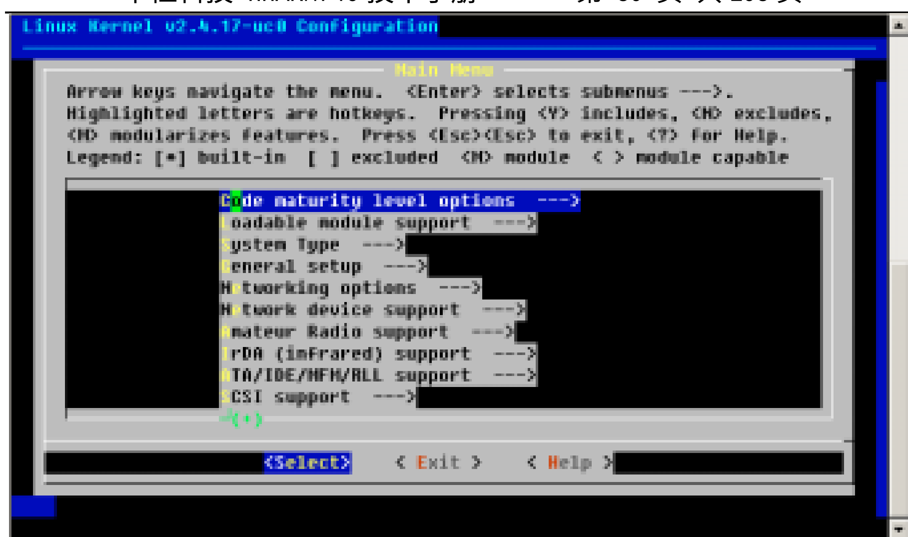
回车后出现主选单，选择[*] Customize Kernel Settings 即表示进行内核配置：

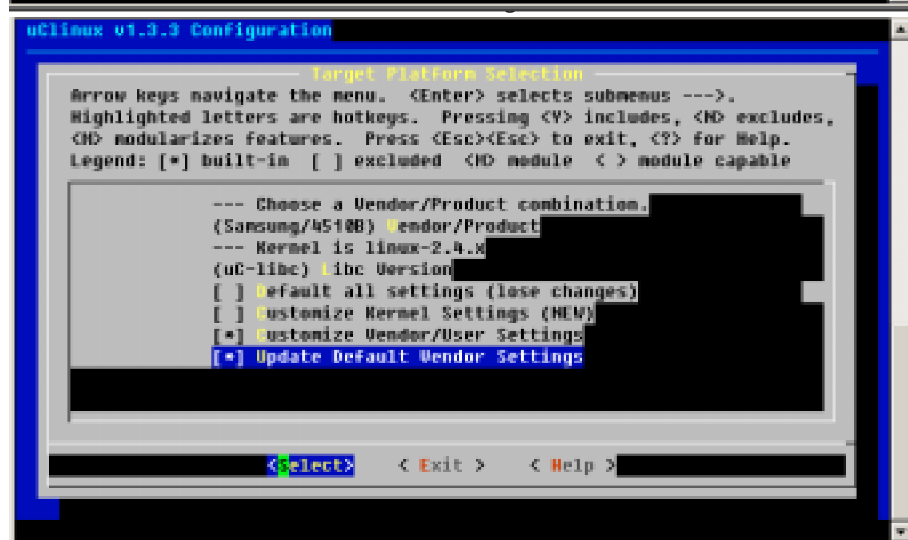


然后选择 Exit，再 Exit，



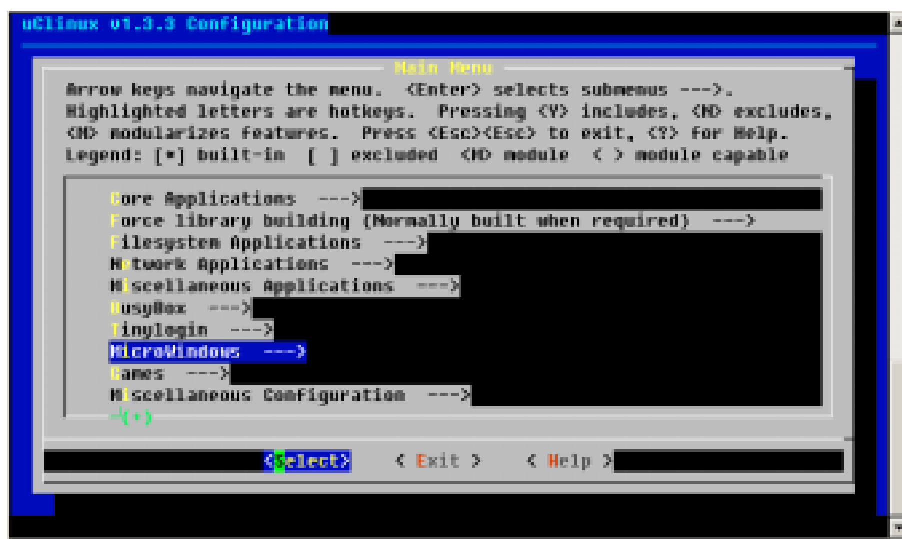
选择 Yes，则退出当前的菜单，进入下面的内核配置主菜单。


Linux Kernel v2.4.17-uc0 Configuration
Main Menu
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help.
Legend: [*] built-in [ ] excluded <M> module < > module capable
Code maturity level options --->
Loadable module support --->
System Type --->
General setup --->
Networking options --->
Network device support --->
Amateur Radio support --->
IrDA (infrared) support --->
ATA/IDE/MFM/RLL support --->
SCSI support --->
<Select> < Exit > < Help >


uClinux v1.3.3 Configuration
Target Platform Selection
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help.
Legend: [*] built-in [ ] excluded <M> module < > module capable
--- Choose a Vendor/Product combination.
(Samsung/4510B) Vendor/Product
--- Kernel is linux-2.4.x
(uC-libc) Libc Version
[ ] Default all settings (lose changes)
[ ] Customize Kernel Settings (NEW)
[*] Customize Vendor/User Settings
[*] Update Default Vendor Settings
<Select> < Exit > < Help >


Do you wish to save your new kernel configuration?
< Yes > < No >

选择 Yes，则退出当前的菜单，进入下面的应用配置主菜单。这里用户可



以任意添加/删除自己的应用程序集。

例如添加 Microwindows 应用软件，则选中 MicroWindows 回车，进入其选择配置菜单，选项如下所示：

[*] MicroWindows

　　--- Compiling Options

　　[*] Optimize

　　[ ] Debug

　　[ ] Verbose

　--- Libraries

　[*] Microwin

　[*] NanoX

　[ ] SharedLibs

　[ ] NWidget

[ ] OBJFramework

--- Demos

[*] MicrowinDemo

[*] NanoXDemo

--- Applications

[*] NanoWM

--- Settings

(CONFIG_USER_MICROWIN_MWPF_TRUECOLOR0888) Screeen PixType

[ ] Link App into server

[ ] Have File IO

[*] Have BMP Support

[*] Have GIF Support

[ ] Have PNM Support

[ ] Have XPM Support

[ ] Have JPEG Support

[ ] Have PNG Support

[ ] Have T1LIB Support

[ ] Have FreeType Support

[ ] Have SharedMemory Support

[ ] Have Han Zi Ku font Support

[ ] Have Big5 Support

[ ] Have GB2312 Support

[ ] Have MS Fonts

[ ] Build Screen Driver only

[ ] Window Erase Move

[ ] Window UpdateRegions Move

[ ] Gray Palette

--- Display Config
[ ] X11 Display
[*] Frame Buffer Display
[ ] Frame Buffer VGA
[ ] VT Switch
[ ] Portrait Mode
[ ] Frame Buffer Reverse
[ ] VGA Lib
[ ] HW VGA
[ ] Cleopatra VGA
--- Mouse/Touch Screen
[ ] GPM Mouse
  [ ] Serial Mouse
  [ ] TP Mouse
  [ ] TP Helio
  [ ] ADS Mouse
  [ ] IPAQ Mouse
  [ ] Harrier Mouse
  [ ] PSION Mouse
  [ ] uClinux/Palm TS
  [ ] Cleopatra Mouse
  [*] No Mouse
--- Keyboard
  [ ] TTY Keyboard
  [ ] Scan Keyboard
  [ ] Pipe Keyboard
  [ ] Cleopatra Keyboard
  [*] No Keyboard

--- Install These Applications
[ ] Landmine
[ ] Launcher
[ ] Move
[ ] MTerm
[ ] MTest
[ ] MTest2
[ ] MUserFd
[*] Nano-X
[*] NanoWM
[ ] NPanel
[ ] NTerm
[ ] NTest
[ ] NTetris
[ ] NXclock
[ ] NXev
[ ] NXkbd
[ ] NXlsclients
[ ] NXterm
[ ] NXView
[ ] Slider
[ ] VNC
[ ] World

选择完毕后，退出并保存配置，执行 make 重新编译烧写则在板子上/bin/目录下就可以看到 Microwin 的应用程序 nano-X、nanowm 和 npanel、nxclock 等演示程序。其中 nano-X 为 X-Server，nanowm 相当于客户端 SHELL，而 npanel 等小程序则为在 SHELL 下执行的应用程序。执行顺序为：nano-X&，然后 nanowm&，最后执行各个小的演示程序，如 npanel 等。

整个编译过程简介：

在 uClinux 目录下执行 make 即可完成所有编译工作，在 images 目录下生成 zImage 和 romfs.img 两个文件．编译过程总体上完成两个工作：

编译内核：

即编译 linux 目录。生成 linux 压缩内核 linux.zip。

编译应用程序及根文件系统：

先编译应用程序所用的 libc 库，即 uClinux/lib 目录下内容；然后再根据 user/Makefile 中 DIRSy 所指定的要编译应用程序目录链表逐一进行编译，并将各个可执行文件复制到 uClinux/romfs/bin/目录下，以供打包烧写。最后利用 genromfs 生成根文件系统镜像 romfs.img。

整个 make 过程由 uClinux/Makefile 文件指定并控制， 执行 make 就是执行这个 Makefile 中指定的操作。Makefile 是由编译器（gcc）解释执行的，它的语法是 gcc 可识别的。Makefile 的使用是为了简化编译过程，它本身可以看作是一个批处理过程，使得编译器可以连续完成对大量 C 代码文件的编译和链接而不需要认为的参与。例如，在 PC LINUX 上要手工编译一个 hello.c 文件，并生成 hello 的可执行程序，只要手工输入执行：

gcc –o hello hello.c

即可。但若要有数千个.c 文件需要编译，这样手工的命令式编译显然是不可行的，因此就有了 Makefile 的需求，它把所有的编译链接命令写成一个文件，由编译器自动调用执行。

uClinux/目录下的这个 Makefile 是个总领式的文件，通过它又层层包含调用各个目录、子目录下面对应的 Makefile，就这样层层调用下去，从而完成整个软件系统的编译。

下面根据 uClinux/Makefile 文件的内容（内容有删节）大致介绍一下整个编译的调用关系，借此用户可以了解一下内核是如何编译的，自己的应用是如何被编译的，以及最终的 IMAGE 文件是如何生成的。

include .config

#all 告诉编译器执行 make 都要分完成哪些工作步骤，这是最重要的地方！

```
#看一个 Makefile 首先要从它的 all 看起，就相当于 C 语言的 main()函数
#的作用。然后再从 all 层层分析下去。
all: subdirs romfs modules modules_install image
```

这里共四步工作，即：

```
subdirs
romfs
modules
modules_install
image
```

下面将在 Makefile 中分别找到各步工作对应的部分：

```
#下面是一些路径相关的宏定义，它们都被 exported 从而可以作为全局变
#量而被其它 Makefile 所用。
# Makefile 中用$(变量名)来使用变量，这中用法对许多熟悉各种脚本语
#言的用户并不陌生。
LINUXDIR = $(CONFIG_LINUXDIR)
LIBCDIR  = $(CONFIG_LIBCDIR)
ROOTDIR  = $(shell pwd)
PATH := $(PATH):$(ROOTDIR)/tools
HOSTCC   = unset GCC_EXEC_PREFIX; cc
IMAGEDIR = $(ROOTDIR)/images
ROMFSDIR = $(ROOTDIR)/romfs
ROMFSINST= romfs-inst.sh
SCRIPTSDIR = $(ROOTDIR)/config/scripts
TFTPDIR    = /tftpboot
LINUX_CONFIG  = $(ROOTDIR)/$(LINUXDIR)/.config
CONFIG_CONFIG = $(ROOTDIR)/config/.config
MODULES_CONFIG = $(ROOTDIR)/modules/.config
```

```
CONFIG_SHELL := $(shell if [ -x "$$BASH" ]; then echo $$BASH; \
      else if [ -x /bin/bash ]; then echo /bin/bash; \
      else echo sh; fi ; fi)

ifeq (config.arch,$(wildcard config.arch))
include config.arch
ARCH_CONFIG = $(ROOTDIR)/config.arch
export ARCH_CONFIG
endif

DIRS    = $(VENDOR_TOPDIRS) lib user
#在这里$(VENDOR_TOPDIRS)变量没有定义，可以认为为空
VENDDIR = $(ROOTDIR)/vendors/$(CONFIG_VENDOR)/$(CONFIG_PRODUCT)/.

#输出如下宏作为全局变量
export VENDOR PRODUCT ROOTDIR LINUXDIR HOSTCC CONFIG_SHELL
export CONFIG_CONFIG LINUX_CONFIG ROMFSDIR SCRIPTSDIR
export VERSIONPKG VERSIONSTR ROMFSINST PATH IMAGEDIR TFTPDIR

#下面是对 make config，make menuconfig，make xconfig 的处理
#对我们的编译过程没有用处，可以不理会。
.PHONY: config.tk config.in
config.in:
    config/mkconfig > config.in

config.tk: config.in
    $(MAKE) -C $(SCRIPTSDIR) tkparse
    ARCH=dummy $(SCRIPTSDIR)/tkparse < config.in > config.tmp
```

```
    @if [ -f /usr/local/bin/wish ]; then \
        echo '#!'"/usr/local/bin/wish -f" > config.tk; \
    else \
        echo '#!'"/usr/bin/wish -f" > config.tk; \
    fi
    cat $(SCRIPTSDIR)/header.tk >> ./config.tk
    cat config.tmp >> config.tk
    rm -f config.tmp
    echo "set defaults \"/dev/null\"" >> config.tk
    echo "set help_file \"config/Configure.help\"" >> config.tk
    cat $(SCRIPTSDIR)/tail.tk >> config.tk
    chmod 755 config.tk

.PHONY: xconfig
xconfig: config.tk
    @wish -f config.tk
    @config/setconfig defaults
    @if egrep "^CONFIG_DEFAULTS_KERNEL=y" .config > /dev/null; then
\
        $(MAKE) linux_xconfig; \
     fi
    @if egrep "^CONFIG_DEFAULTS_MODULES=y" .config > /dev/null;
then \
        $(MAKE) modules_xconfig; \
     fi
    @if egrep "^CONFIG_DEFAULTS_VENDOR=y" .config > /dev/null; then
\
        $(MAKE) config_xconfig; \
```

```
     fi
    @config/setconfig final

.PHONY: config
config: config.in
    @HELP_FILE=config/Configure.help \
        $(CONFIG_SHELL) $(SCRIPTSDIR)/Configure config.in
    @config/setconfig defaults
    @if egrep "^CONFIG_DEFAULTS_KERNEL=y" .config > /dev/null; then
\
        $(MAKE) linux_config; \
     fi
    @if  egrep  "^CONFIG_DEFAULTS_MODULES=y"  .config  >  /dev/null;
then \
        $(MAKE) modules_config; \
     fi
    @if egrep "^CONFIG_DEFAULTS_VENDOR=y" .config > /dev/null; then
\
        $(MAKE) config_config; \
     fi
    @config/setconfig final
.PHONY: menuconfig
menuconfig: config.in
    $(MAKE) -C $(SCRIPTSDIR)/lxdialog all
    @HELP_FILE=config/Configure.help \
        $(CONFIG_SHELL) $(SCRIPTSDIR)/Menuconfig config.in
    @config/setconfig defaults
    @if egrep "^CONFIG_DEFAULTS_KERNEL=y" .config > /dev/null; then
```

```
\
        $(MAKE) linux_menuconfig; \
     fi
    @if egrep "^CONFIG_DEFAULTS_MODULES=y" .config > /dev/null; then \
        $(MAKE) modules_menuconfig; \
     fi
    @if egrep "^CONFIG_DEFAULTS_VENDOR=y" .config > /dev/null; then \
        $(MAKE) config_menuconfig; \
     fi
    @config/setconfig final

.PHONY: oldconfig
oldconfig:
    @$(MAKE) oldconfig_linux
    @$(MAKE) oldconfig_modules
    @$(MAKE) oldconfig_config
    @config/setconfig final
####################################################################
####################################################################
#下面是编译的工作之一
modules
.PHONY: modules
modules:
    . $(LINUXDIR)/.config; if [ "$$CONFIG_MODULES" = "y" ]; then \
        [ -d $(LINUXDIR)/modules ] || mkdir $(LINUXDIR)/modules; \
        make -C $(LINUXDIR) modules; \
```

```
    fi
#下面是编译工作之一
modules_install
.PHONY: modules_install
modules_install:
    . $(LINUXDIR)/.config; if [ "$$CONFIG_MODULES" = "y" ]; then \
        [ -d $(ROMFSDIR)/lib/modules ] || mkdir -p $(ROMFSDIR)/lib/modules; \
        make -C $(LINUXDIR) INSTALL_MOD_PATH=$(ROMFSDIR) DEPMOD=true modules_install; \
        rm -f $(ROMFSDIR)/lib/modules/*/build; \
        find $(ROMFSDIR)/lib/modules -type f | xargs -r $(STRIP) -g; \
    fi
```

以上两步工作都是针对支持动态 MODULES 的情况，我们目前采用的 uClinux 都没有采用 MODULES 模块，因此上面两步工作都没有用。

```
################################################################
################################################################
#下面是对在 linux-2.4.x 下 make config，make xconfig，make menuconfig
#的响应处理。
linux_xconfig:
    $(MAKE) -C $(LINUXDIR) xconfig
linux_menuconfig:
    $(MAKE) -C $(LINUXDIR) menuconfig
linux_config:
    $(MAKE) -C $(LINUXDIR) config
modules_xconfig:
    [ ! -d modules ] || $(MAKE) -C modules xconfig
```

```
modules_menuconfig:
    [ ! -d modules ] || $(MAKE) -C modules menuconfig
modules_config:
    [ ! -d modules ] || $(MAKE) -C modules config
config_xconfig:
    $(MAKE) -C config xconfig
config_menuconfig:
    $(MAKE) -C config menuconfig
config_config:
    $(MAKE) -C config config
oldconfig_config:
    $(MAKE) -C config oldconfig
oldconfig_modules:
    [ ! -d modules ] || $(MAKE) -C modules oldconfig
oldconfig_linux:
    $(MAKE) -C $(LINUXDIR) oldconfig
###################################################################
###################################################################
#下面才是我们在 uClinux 执行 make 所做的工作！
romfs
.PHONY: romfs
romfs:
    for dir in $(DIRS) ; do $(MAKE) -C $$dir romfs || exit 1 ; done
    #在 user 下每个应用程序目录下执行 make romfs
    -find $(ROMFSDIR)/. -name CVS | xargs -r rm -rf
image
.PHONY: image
image:
```

```
    [ -d $(IMAGEDIR) ] || mkdir $(IMAGEDIR)
    $(MAKE) -C $(VENDDIR) image
    #在 uClinux/vendors/Samsung/740 目录下执行 make image（后面详述）

.PHONY: linux
linux:
    @if [ ! -f $(LINUXDIR)/.depend ] ; then \
    #【若找不到 linux-2.4.x/.depend 文件，就提示需要 make config】
        echo "ERROR: you need to do a 'make dep' first" ; \
        exit 1 ; \
    fi
    $(MAKE) -C $(LINUXDIR) $(LINUXTARGET) || exit 1

subdirs
.PHONY: subdirs
subdirs: linux
    for dir in $(DIRS) ; do $(MAKE) -C $$dir || exit 1 ; done
```

【上面这一句虽短，但确是最主要的工作所在，首先它先依赖于 linux 节工作的完成；又注意到前面定义了：

DIRS = lib user

因此这一句：

make -C $$dir

就完成了对内核（linux 目录）的编译、对 libC 库（lib 目录）的编译、对所有应用程序（user 下所有指定要编译的目录）的编译。

make -C $$dir 就调用对应那个目录下的 Makefile，即分别是 linux-2.4.x/Makefile，lib/Makefile 和 user/Makefile，这些 Makefile 又层层包含调用下面的各个目录的 Makefile，从而完成整个编译过程。】

```
#################################################################
#################################################################
#对 make dep 的处理
dep:
    @if [ ! -f $(LINUXDIR)/.config ] ; then \
    #【若找不到 linux-2.4.x/.config 文件，就提示需要 make config】
echo "ERROR: you need to do a 'make config' first" ; \
        exit 1 ; \
    fi
    $(MAKE) -C $(LINUXDIR) dep

#对 make clean 的处理
clean:
    for dir in $(LINUXDIR) $(DIRS); do $(MAKE) -C $$dir clean ; done
    rm -rf $(ROMFSDIR)/*
    rm -f $(IMAGEDIR)/*
    rm -f config.tk

#错误提示
config_error:
    @echo "*****************************************************"
    @echo "You have not run make config."
    @echo "The build sequence for this source tree is:"
    @echo "1. 'make config' or 'make xconfig'"
    @echo "2. 'make dep'"
    @echo "3. 'make'"
    @echo "*****************************************************"
    @exit 1
```

【注意】
Makefile 中使用了许多.PHONY:节，它并没有什么作用，只是告知编译器，它的:号后面并不是一个文件。例如：
.PHONY: romfs　（没有这句也可以，但是若目录下有一个文件名字叫做romfs，则会报错 make: ` ‘romfs’ is up to date。加了 PHONY 就是通知编译器，不要把 romfs 看作一个目标文件。）
romfs:
　……
Makefile 还提供了许多独立的目标，可以直接用 make 命令指定目标单独编译。例如：可以直接 make dep，make romfs 执行。

由这个总领的 Makefile 的结构可见，通常的我们在 uClinux 下执行 make 主要做的工作按顺序步骤如下：
编译 Linux 内核
编译 libc 函数库
编译 user 下所有应用程序
make romfs
make image

【注意】
在 linux-2.4.x,lib 和 user 下的 Makefile 中都用到了类似 CROSS_COMPILE，CFLAGS，LDFLAGS 等等这样的常用的公用的宏，它们是在 **uClinux/vendors/Winbond/W90N740/config.arch** 中定义的。这个文件给出了编译 lib 下 libc 库文件和 user 下应用程序的许多公用参数，其中最主要的就是编译参数：CFLAGS 和链接参数：LDFLAGS。而内核中所用的编译参数则是在它自己的 Makefile 中定义的，而 linux-2.4.x 目录下的

Rules.make 则定义了.c 和.S 等文件的编译规则。

下面给出 **uClinux/vendors/Winbod/W90N740/config.arch** 文件内容（有删节）：

```
CONSOLE_BAUD_RATE = 115200
#告诉编译器该如何行为的统一参数和宏的定义之处
MACHINE          = arm
ARCH             = armnommu
CROSS_COMPILE    = arm-elf-
CROSS            = $(CROSS_COMPILE)

#C 编译器
CC        = $(CROSS_COMPILE)gcc #汇编编译器
AS        = $(CROSS_COMPILE)as
#C++语言编译器
CXX       = $(CROSS_COMPILE)g++
AR        = $(CROSS_COMPILE)ar
#链接器
LD        = $(CROSS_COMPILE)ld
OBJCOPY   = $(CROSS_COMPILE)objcopy
RANLIB    = $(CROSS_COMPILE)ranlib
#可执行文件格式转换器 ELF→FLAT
ELF2FLT   = elf2flt
STRIPTOOL = $(CROSS_COMPILE)strip
STRIP     = $(STRIPTOOL)
#下面是编译 lib 库时的编译参数和宏定义
ifdef UCLINUX_BUILD_LIB
  ifdef CONFIG_LIB_DEBUG
    CFLAGS  :=  -O1 -g
```

```
   else
      CFLAGS   :=   -O2 -g -fomit-frame-pointer
   endif
      CFLAGS   += $(VENDOR_CFLAGS)
      CFLAGS   += -fno-builtin
CFLAGS   += -DEMBED

      # don't want all the CFLAGS for uClibc/Config
ARCH_CFLAGS = $(CFLAGS)
      CFLAGS      +=    -I$(ROOTDIR)/lib/$(CONFIG_LIBCDIR)/include    -
I$(ROOTDIR)
      CFLAGS   += -Dlinux -D__linux__ -D__uClinux__ -Dunix

      # the following is needed for uClinux-2.4
      CFLAGS   += -I$(ROOTDIR)/$(LINUXDIR)/include

      LDFLAGS   = $(CFLAGS) -Wl,-elf2flt

      UCLINUX_BUILD_SET=1
endif

#下面是编译 user 下所有应用程序时的编译参数和宏定义
ifdef UCLINUX_BUILD_USER
      GCC_DIR = $(shell $(CC) -v 2>&1|grep specs|sed -e 's/.* \(.*\)specs/\1\./')
      GCC_LIB = /usr/local/arm-elf/lib

            LIBC           = -lc
            LIBM            = -lm
```

```
    LIBNET        = -lnet
    LIBDES        = -ldes
    LIBPCAP       = -lpcap
    LIBSSL        = -lssl
    LIBCRYPTO     = -lcrypto
    LIBCRYPT      = -lcrypt
    LIBGCC        = -lgcc
    LIBIBERTY     = -liberty
    LIBIO         = -lio
    LIBIOSTREAM = -liostream
    LIBSTDCPP     = -lstdc++

LDPATH = \
        -L$(ROOTDIR)/lib/$(LIBCDIR)/. \
        -L$(ROOTDIR)/lib/$(LIBCDIR)/lib \
        -L$(ROOTDIR)/lib/libm \
        -L$(ROOTDIR)/lib/libnet \
        -L$(ROOTDIR)/lib/libdes \
        -L$(ROOTDIR)/lib/libpcap \
        -L$(ROOTDIR)/lib/libssl

INCLIBC = -I$(ROOTDIR)/lib/$(CONFIG_LIBCDIR)/include
INCLIBM = -I$(ROOTDIR)/lib/libm
INCNET    = -I$(ROOTDIR)/lib/libnet
INCDES    = -I$(ROOTDIR)/freeswan/libdes
INCGMP    = -I$(ROOTDIR)/lib/libgmp
INCPCAP = -I$(ROOTDIR)/lib/libpcap
INCSSL    = -I$(ROOTDIR)/lib/libssl/include
```

```
  INCVEND = -I$(ROOTDIR)/vendors/include

ifdef CONFIG_USER_DEBUG
  CFLAGS   :=   -O1 -g
else
  CFLAGS   :=   -Os -g -fomit-frame-pointer
endif
  CFLAGS   += $(VENDOR_CFLAGS)
  CFLAGS   += -Dlinux -D__linux__ -Dunix -D__uClinux__ -DEMBED
  CFLAGS   += $(INCLIBC) $(INCLIBM)
  CFLAGS   += -I$(ROOTDIR)
  CFLAGS   += -fno-builtin
  CFLAGS   += -msep-data
  #
  # the following is needed for 2.4
  #
  CFLAGS   += -I$(ROOTDIR)/$(LINUXDIR)/include

  CXXFLAGS = $(CFLAGS) $(INCCXX) -fname-mangling-version-0

  LDFLAGS   = $(CFLAGS) -Wl,-elf2flt
  LDLIBS     = $(LDPATH) $(LIBC)
  CXXLIBS   = $(LDPATH) $(LIBSTDCPP) $(LIBIOSTREAM) $(LIBIO)
$(LIBIBERTY) \
                    $(LIBC) $(LIBGCC)

  FLTFLAGS :=
  export FLTFLAGS
```

```
        # for anyone still using it
        CONVERT = /bin/true

        UCLINUX_BUILD_SET=1
Endif
```

内核，库和应用程序三者中和我们关系最密切的就是应用程序了，即 user 目录，因此专门看看 user 下的 Makefile。（有删节）

记住：看一个 Makefile 首先要找其入口：**all**

.EXPORT_ALL_VARIABLES:　　　　【输出所有的宏作为全局变量】

```
ifndef ROOTDIR
ROOTDIR=..
endif

UCLINUX_BUILD_USER=1
include $(LINUX_CONFIG)
include $(CONFIG_CONFIG)
```

include $(ARCH_CONFIG)

#即：**include uClinux/vendors/Samsung/740/config.arch**

```
-include $(MODULES_CONFIG)

VEND=$(ROOTDIR)/vendors
```

dir_y = $(VEND)/$(CONFIG_VENDOR)/$(CONFIG_PRODUCT)/.

#注意：user 下的编译目录链表 dir_y 中第一项就是：

#uClinux/vendors/**Samsung/740/**，因此这个目录也要参与编译，并参与后

```
#面的 make romfs
dir_n =
dir_    =
dir_y += memtools

dir_$(CONFIG_JFFS_FS)                          += mtd-utils
dir_$(CONFIG_JFFS2_FS)                         += mtd-utils

dir_$(CONFIG_USER_AGETTY_AGETTY)               += agetty
dir_$(CONFIG_USER_AT_AT)                       += at
dir_$(CONFIG_USER_AT_ATD)                      += at
dir_$(CONFIG_USER_AT_ATRUN)                    += at
dir_$(CONFIG_USER_BASH_BASH)                   += bash
dir_$(CONFIG_USER_BOA_SRC_BOA)                 += boa
dir_$(CONFIG_USER_BOOTTOOLS_FLASHLOADER)       += boottools
dir_$(CONFIG_USER_BOOTTOOLS_HIMEMLOADER)       += boottools
dir_$(CONFIG_USER_BPALOGIN_BPALOGIN)           += bpalogin
dir_$(CONFIG_USER_BRCFG_BRCFG)                 += brcfg
dir_$(CONFIG_USER_BUSYBOX_BUSYBOX)             += busybox
dir_$(CONFIG_USER_CAL_CAL)                     += cal
dir_$(CONFIG_USER_CAL_DATE)                    += cal
dir_$(CONFIG_USER_CGI_GENERIC)                 += cgi_generic
dir_$(CONFIG_USER_CHAT_CHAT)                   += pppd/chat
dir_$(CONFIG_USER_CKSUM_CKSUM)                 += cksum
dir_$(CONFIG_USER_CLOCK_CLOCK)                 += clock
dir_$(CONFIG_USER_CPU_CPU)                     += cpu
dir_$(CONFIG_USER_CRON_CRON)                   += cron
dir_$(CONFIG_USER_DHRYSTONE_DHRYSTONE)         += dhrystone
```

```
dir_$(CONFIG_USER_DHCP_ISC_SERVER_DHCPD)       += dhcp-isc
dir_$(CONFIG_USER_DHCPCD_DHCPCD)               += dhcpcd
dir_$(CONFIG_USER_DHCPCD_NEW_DHCPCD)           += dhcpcd-new
dir_$(CONFIG_USER_DHCPD_DHCPD)                 += dhcpd
dir_$(CONFIG_USER_DIALD_DIALD)                 += diald
dir_$(CONFIG_USER_DISCARD_DISCARD)             += discard
dir_$(CONFIG_USER_DNSMASQ_DNSMASQ)             += dnsmasq
dir_$(CONFIG_USER_E2FSPROGS_E2FSCK_E2FSCK)    += e2fsprogs
dir_$(CONFIG_USER_E2FSPROGS_MISC_BADBLOCKS) += e2fsprogs
dir_$(CONFIG_USER_E2FSPROGS_MISC_CHATTR)       += e2fsprogs
dir_$(CONFIG_USER_E2FSPROGS_MISC_DUMPE2FS)    += e2fsprogs
dir_$(CONFIG_USER_E2FSPROGS_MISC_CHATTR)       += e2fsprogs
dir_$(CONFIG_USER_E2FSPROGS_MISC_DUMPE2FS)    += e2fsprogs
dir_$(CONFIG_USER_E2FSPROGS_MISC_E2LABEL)      += e2fsprogs
dir_$(CONFIG_USER_E2FSPROGS_MISC_FSCK)         += e2fsprogs
dir_$(CONFIG_USER_E2FSPROGS_MISC_LSATTR)       += e2fsprogs
dir_$(CONFIG_USER_E2FSPROGS_MISC_MKE2FS)       += e2fsprogs
dir_$(CONFIG_USER_E2FSPROGS_MISC_MKLOST_FOUND)                +=
e2fsprogs
dir_$(CONFIG_USER_E2FSPROGS_MISC_TUNE2FS)      += e2fsprogs
dir_$(CONFIG_USER_E2FSPROGS_MISC_UUIDGEN)      += e2fsprogs
dir_$(CONFIG_USER_ELVISTINY_VI)                += elvis-tiny
dir_$(CONFIG_USER_ETHATTACH_ETHATTACH)         += ethattach
dir_$(CONFIG_USER_EZIPUPDATE_EZIPUPDATE)       += ez-ipupdate
dir_$(CONFIG_USER_FDISK_FDISK)                 += fdisk
dir_$(CONFIG_USER_FILEUTILS_CAT)               += fileutils
dir_$(CONFIG_USER_FILEUTILS_CHGRP)             += fileutils
dir_$(CONFIG_USER_FILEUTILS_CHMOD)             += fileutils
```

```
dir_$(CONFIG_USER_FILEUTILS_CHOWN)              += fileutils
dir_$(CONFIG_USER_FILEUTILS_CMP)                += fileutils
dir_$(CONFIG_USER_FILEUTILS_CP)                 += fileutils
dir_$(CONFIG_USER_FILEUTILS_DD)                 += fileutils
dir_$(CONFIG_USER_FILEUTILS_GREP)               += fileutils
dir_$(CONFIG_USER_FILEUTILS_L)                  += fileutils
dir_$(CONFIG_USER_FILEUTILS_LN)                 += fileutils
dir_$(CONFIG_USER_FILEUTILS_LS)                 += fileutils
dir_$(CONFIG_USER_FILEUTILS_MKDIR)              += fileutils
dir_$(CONFIG_USER_FILEUTILS_MKFIFO)             += fileutils
dir_$(CONFIG_USER_FILEUTILS_MKNOD)              += fileutils
dir_$(CONFIG_USER_FILEUTILS_MORE)               += fileutils
dir_$(CONFIG_USER_FILEUTILS_MV)                 += fileutils
dir_$(CONFIG_USER_FILEUTILS_RM)                 += fileutils
dir_$(CONFIG_USER_FILEUTILS_RMDIR)              += fileutils
dir_$(CONFIG_USER_FILEUTILS_SYNC)               += fileutils
dir_$(CONFIG_USER_FILEUTILS_TOUCH)              += fileutils
dir_$(CONFIG_USER_FILEUTILS_SYNC)               += fileutils
dir_$(CONFIG_USER_FILEUTILS_TOUCH)              += fileutils
dir_$(CONFIG_USER_FLASHW_FLASHW)                += flashw
dir_$(CONFIG_USER_FLATFSD_FLATFSD)              += flatfsd
dir_$(CONFIG_USER_FREESWAN)                     += freeswan
dir_$(CONFIG_USER_FROB_LED_FROB_LED)            += frob-led
dir_$(CONFIG_USER_FTP_FTP_FTP)                  += ftp
dir_$(CONFIG_USER_GDBSERVER_GDBREPLAY)          += gdbserver
dir_$(CONFIG_USER_GDBSERVER_GDBSERVER)          += gdbserver
dir_$(CONFIG_USER_GETTYD_GETTYD)                += gettyd
dir_$(CONFIG_USER_HD_HD)                        += hd
```

```
dir_$(CONFIG_USER_HTTPD_HTTPD)                  += httpd
dir_$(CONFIG_USER_HWCLOCK_HWCLOCK)              += hwclock
dir_$(CONFIG_USER_IFATTACH_IFATTACH)            += ifattach
dir_$(CONFIG_USER_INETD_INETD)                  += inetd
dir_$(CONFIG_USER_INIT.ORG_INIT)                += init.org
dir_$(CONFIG_USER_INIT_EXPAND)                  += init
dir_$(CONFIG_USER_INIT_INIT)                    += init
dir_$(CONFIG_USER_IPCHAINS_IPCHAINS)            += ipchains
dir_$(CONFIG_USER_IPFWADM_IPFWADM)              += ipfwadm
dir_$(CONFIG_USER_IPMASQADM_IPMASQADM)          += ipmasqadm
dir_$(CONFIG_USER_IPPORTFW_IPPORTFW)            += ipportfw
dir_$(CONFIG_USER_IPREDIR_IPREDIR)              += ipredir
dir_$(CONFIG_USER_IPROUTE2)                     += iproute2
dir_$(CONFIG_USER_IPTABLES_IPTABLES)            += iptables
dir_$(CONFIG_USER_KLAXON_KLAXON)                += klaxon
dir_$(CONFIG_USER_LANG_A60)                     += a60
dir_$(CONFIG_USER_LCD_LCD)                      += lcd
dir_$(CONFIG_USER_LEVEE_VI)                     += levee
dir_$(CONFIG_USER_LIRC)                         += lirc
dir_$(CONFIG_USER_LISSA_LISSA)                  += lissa
dir_$(CONFIG_USER_LOATTACH_LOATTACH)            += loattach
dir_$(CONFIG_USER_LOGIN_LOGIN)                  += login
dir_$(CONFIG_USER_LOATTACH_LOATTACH)            += loattach
dir_$(CONFIG_USER_LOGIN_LOGIN)                  += login
dir_$(CONFIG_USER_LOGIN_PASSWD)                 += login
dir_$(CONFIG_USER_MAIL_MAIL_IP)                 += mail
dir_$(CONFIG_USER_MATH_TEST)                    += mathtest
dir_$(CONFIG_USER_MAWK_AWK)                     += mawk
```

```
dir_$(CONFIG_USER_MTDUTILS)                    += mtd-utils
dir_$(CONFIG_USER_MICROWIN)                    += microwin
dir_$(CONFIG_USER_MINI_HTTPD_MINI_HTTPD)       += mini_httpd
dir_$(CONFIG_USER_MOUNT_MOUNT)                 += mount
dir_$(CONFIG_USER_MOUNT_UMOUNT)                += mount
dir_$(CONFIG_USER_MP3PLAY_MP3PLAY)             += mp3play
dir_$(CONFIG_USER_MSNTP_MSNTP)                 += msntp
dir_$(CONFIG_USER_MUSICBOX_MUSICBOX)           += musicbox
dir_$(CONFIG_USER_NETFLASH_NETFLASH)           += netflash
dir_$(CONFIG_USER_NET_TOOLS_ARP)               += net-tools
dir_$(CONFIG_USER_NET_TOOLS_HOSTNAME)          += net-tools
dir_$(CONFIG_USER_NET_TOOLS_IFCONFIG)          += net-tools
dir_$(CONFIG_USER_NET_TOOLS_NAMEIF)            += net-tools
dir_$(CONFIG_USER_NET_TOOLS_NETSTAT)           += net-tools
dir_$(CONFIG_USER_NET_TOOLS_PLIPCONFIG)        += net-tools
dir_$(CONFIG_USER_NET_TOOLS_RARP)              += net-tools
dir_$(CONFIG_USER_NET_TOOLS_ROUTE)             += net-tools
dir_$(CONFIG_USER_NET_TOOLS_SLATTACH)          += net-tools
dir_$(CONFIG_USER_NET_TOOLS_IPMADDR)           += net-tools
dir_$(CONFIG_USER_NET_TOOLS_IPTUNNEL)          += net-tools
dir_$(CONFIG_USER_NET_TOOLS_MII_TOOL)          += net-tools
dir_$(CONFIG_USER_NULL_NULL)                   += null
dir_$(CONFIG_USER_NWSH_NWSH)                   += nwsh
dir_$(CONFIG_USER_PALMBOT_PALMBOT)             += palmbot
dir_$(CONFIG_USER_PCIUTILS_LSPCI)              += pciutils
dir_$(CONFIG_USER_PCIUTILS_SETPCI)             += pciutils
dir_$(CONFIG_USER_PCMCIA_CS)                   += pcmcia-cs
dir_$(CONFIG_USER_PCIUTILS_SETPCI)             += pciutils
```

```
dir_$(CONFIG_USER_PCMCIA_CS)                    += pcmcia-cs
dir_$(CONFIG_USER_PERL_PERL)                    += perl
dir_$(CONFIG_USER_PING_PING)                    += ping
dir_$(CONFIG_USER_PLAY_PLAY)                    += play
dir_$(CONFIG_USER_PLAY_TONE)                    += play
dir_$(CONFIG_USER_PLUG_PLUG)                    += plug
dir_$(CONFIG_USER_PORTMAP_PORTMAP)              += portmap
dir_$(CONFIG_USER_PPPD_PPPD_PPPD)               += pppd
dir_$(CONFIG_USER_PPTP_CLIENT_PPTP)             += pptp-client
dir_$(CONFIG_USER_PPTP_CLIENT_PPTP_CALLMGR) += pptp-client
dir_$(CONFIG_USER_PPTP_CLIENT_PPTP_CALLMGR) += pptp-client
dir_$(CONFIG_USER_PPTPD_PPTPCTRL)               += pptpd
dir_$(CONFIG_USER_PPTPD_PPTPD)                  += pptpd
dir_$(CONFIG_USER_PROCPS_FREE)                  += procps
dir_$(CONFIG_USER_PROCPS_KILL)                  += procps
dir_$(CONFIG_USER_PROCPS_PGREP)                 += procps
dir_$(CONFIG_USER_PROCPS_PKILL)                 += procps
dir_$(CONFIG_USER_PROCPS_PS)                    += procps
dir_$(CONFIG_USER_PROCPS_SKILL)                 += procps
dir_$(CONFIG_USER_PROCPS_SNICE)                 += procps
dir_$(CONFIG_USER_PROCPS_SYSCTL)                += procps
dir_$(CONFIG_USER_PROCPS_TLOAD)                 += procps
dir_$(CONFIG_USER_PROCPS_TOP)                   += procps
dir_$(CONFIG_USER_PROCPS_UPTIME)                += procps
dir_$(CONFIG_USER_PROCPS_VMSTAT)                += procps
dir_$(CONFIG_USER_PROCPS_W)                     += procps
dir_$(CONFIG_USER_PROCPS_WATCH)                 += procps
dir_$(CONFIG_USER_PYTHON_PYTHON)                += python
```

```
dir_$(CONFIG_USER_RDATE_RDATE)                    += rdate
dir_$(CONFIG_USER_RAMIMAGE_RAMFS64)               += ramimage
dir_$(CONFIG_USER_RAMIMAGE_RAMFS128)              += ramimage
dir_$(CONFIG_USER_RAMIMAGE_RAMFS256)              += ramimage
dir_$(CONFIG_USER_RAMIMAGE_RAMFS128)              += ramimage
dir_$(CONFIG_USER_RAMIMAGE_RAMFS256)              += ramimage
dir_$(CONFIG_USER_READPROFILE_READPROFILE)        += readprofile
dir_$(CONFIG_USER_RECOVER_RECOVER)                += recover
dir_$(CONFIG_USER_REISERFSPROGS)                  += reiserfsprogs
dir_$(CONFIG_USER_ROOTLOADER_ROOTLOADER)          += rootloader
dir_$(CONFIG_USER_ROUTE_ARP)                      += route
dir_$(CONFIG_USER_ROUTE_IFCONFIG)                 += route
dir_$(CONFIG_USER_ROUTE_MIITOOL)                  += route
dir_$(CONFIG_USER_ROUTE_ROUTE)                    += route
dir_$(CONFIG_USER_ROUTE_NETSTAT)                  += route
dir_$(CONFIG_USER_ROUTED_ROUTED)                  += routed
dir_$(CONFIG_USER_RP_PPPOE_PPPOE)                 += rp-pppoe
dir_$(CONFIG_USER_RTC_M41T11)                     += rtc-m41t11
dir_$(CONFIG_USER_RTC_DS1302)                     += rtc-ds1302
dir_$(CONFIG_USER_SAMBA_SMBD)                     += samba
dir_$(CONFIG_USER_SAMBA_NMBD)                     += samba
dir_$(CONFIG_USER_SAMBA_SMBMOUNT)                 += samba
dir_$(CONFIG_USER_SAMBA_SMBUMOUNT)                += samba
dir_$(CONFIG_USER_SASH_REBOOT)                    += sash
dir_$(CONFIG_USER_SASH_SH)                        += sash
dir_$(CONFIG_USER_SASH_SHUTDOWN)                  += sash
dir_$(CONFIG_USER_SASH_SHUTDOWN)                  += sash
dir_$(CONFIG_USER_SETKEY_SETKEY)                  += setkey
```

```
dir_$(CONFIG_USER_SETSERIAL_SETSERIAL)        += setserial
dir_$(CONFIG_USER_SH_SH)                      += sh
dir_$(CONFIG_USER_SHUTILS_BASENAME)           += shutils
dir_$(CONFIG_USER_SHUTILS_DATE)               += shutils
dir_$(CONFIG_USER_SHUTILS_DIRNAME)            += shutils
dir_$(CONFIG_USER_SHUTILS_ECHO)               += shutils
dir_$(CONFIG_USER_SHUTILS_FALSE)              += shutils
dir_$(CONFIG_USER_SHUTILS_LOGNAME)            += shutils
dir_$(CONFIG_USER_SHUTILS_PRINTENV)           += shutils
dir_$(CONFIG_USER_SHUTILS_LOGNAME)            += shutils
dir_$(CONFIG_USER_SHUTILS_PRINTENV)           += shutils
dir_$(CONFIG_USER_SHUTILS_PWD)                += shutils
dir_$(CONFIG_USER_SHUTILS_TRUE)               += shutils
dir_$(CONFIG_USER_SHUTILS_UNAME)              += shutils
dir_$(CONFIG_USER_SHUTILS_WHICH)              += shutils
dir_$(CONFIG_USER_SHUTILS_WHOAMI)             += shutils
dir_$(CONFIG_USER_SHUTILS_YES)                += shutils
dir_$(CONFIG_USER_SLATTACH_SLATTACH)          += slattach
dir_$(CONFIG_USER_SMBMOUNT_SMBMOUNT)          += smbmount
dir_$(CONFIG_USER_SMBMOUNT_SMBUMOUNT)         += smbmount
dir_$(CONFIG_USER_SMTP_SMTPCLIENT)            += smtpclient
dir_$(CONFIG_USER_SNMPD_SNMPD)                += snmpd
dir_$(CONFIG_USER_STUNNEL_STUNNEL)            += stunnel
dir_$(CONFIG_USER_SQUID_SQUID)                += squid
dir_$(CONFIG_USER_SSH_SSHD)                   += ssh
dir_$(CONFIG_USER_SSH_SSH)                    += ssh
dir_$(CONFIG_USER_SSH_SSHKEYGEN)              += ssh
dir_$(CONFIG_USER_STP_STP)                    += stp
```

```
dir_$(CONFIG_USER_STRACE_STRACE)                += strace
dir_$(CONFIG_USER_STTY_STTY)                    += stty
dir_$(CONFIG_USER_SYSUTILS_DF)                  += sysutils
dir_$(CONFIG_USER_SYSUTILS_FREE)                += sysutils
dir_$(CONFIG_USER_SYSUTILS_HOSTNAME)            += sysutils
dir_$(CONFIG_USER_SYSUTILS_KILL)                += sysutils
dir_$(CONFIG_USER_SYSUTILS_PS)                  += sysutils
dir_$(CONFIG_USER_SYSUTILS_REBOOT)              += sysutils
dir_$(CONFIG_USER_SYSUTILS_SHUTDOWN)            += sysutils
dir_$(CONFIG_USER_TCPBLAST_TCPBLAST)            += tcpblast
dir_$(CONFIG_USER_TCPWRAP_TCPD)                 += tcpwrappers
dir_$(CONFIG_USER_TCPDUMP_TCPDUMP)              += tcpdump
dir_$(CONFIG_USER_TCSH_TCSH)                    += tcsh
dir_$(CONFIG_USER_TELNET_TELNET)                += telnet
dir_$(CONFIG_USER_TCSH_TCSH)                    += tcsh
dir_$(CONFIG_USER_TELNET_TELNET)                += telnet
dir_$(CONFIG_USER_TELNETD_TELNETD)              += telnetd
dir_$(CONFIG_USER_TFTP_TFTP)                    += tftp
dir_$(CONFIG_USER_TFTPD_TFTPD)                  += tftpd
dir_$(CONFIG_USER_THTTPD_THTTPD)                += thttpd
dir_$(CONFIG_USER_TINYLOGIN_CRYPT_CRYPT)        += tinylogin
dir_$(CONFIG_USER_TINYLOGIN_TINYLOGIN)          += tinylogin
dir_$(CONFIG_USER_TIP_TIP)                      += tip
dir_$(CONFIG_USER_TIMEPEG_TPT)                  += tpt
dir_$(CONFIG_USER_TRACEROUTE_TRACEROUTE)        += traceroute
dir_$(CONFIG_USER_TRIPWIRE_SIGGEN)              += tripwire
dir_$(CONFIG_USER_TRIPWIRE_TRIPWIRE)            += tripwire
dir_$(CONFIG_USER_UCDSNMP_SNMPD)                += ucdsnmp
```

```
dir_$(CONFIG_USER_VERSION_VERSION)              += version
dir_$(CONFIG_USER_VIXIECRON_CRON)               += vixie-cron
dir_$(CONFIG_USER_VIXIECRON_CRONTAB)            += vixie-cron
dir_$(CONFIG_USER_VPLAY_VPLAY)                  += vplay
dir_$(CONFIG_USER_VPLAY_VREC)                   += vplay
dir_$(CONFIG_USER_VPLAY_MIXER)                  += vplay
dir_$(CONFIG_USER_VPNLED_VPNLED)                += vpnled
dir_$(CONFIG_USER_LEDCON_LEDCON)                += ledcon
dir_$(CONFIG_USER_WGET)                         += wget
dir_$(CONFIG_USER_WIRELESS_TOOLS)               += wireless_tools
dir_$(CONFIG_USER_ZEBRA_BGPD_BGPD)              += zebra
dir_$(CONFIG_USER_ZEBRA_OSPFD_OSPFD)            += zebra
dir_$(CONFIG_USER_ZEBRA_RIPD_RIPD)              += zebra
dir_$(CONFIG_USER_ZEBRA_ZEBRA_ZEBRA)            += zebra

dir_y += games
```

all:

for i in $(dir_y) ; do make -C $$i || exit $? ; done

#这里就是入口所在，也是所有的工作总领处。其实就是逐个编译所有参
#与编译的目录（dir_y 为目录列表）下的应用程序。

```
romfs:
	for i in $(dir_y) ; do    make -C $$i romfs || exit $? ; done

clean:
	-for i in $(dir_y) $(dir_n) $(dir_) ; do \
		[ ! -d $$i ] || make -C $$i clean; \
```

done

【注】

uClinux 唯一支持的可执行文件格式就是 FLAT 格式。其它都不支持。所以所有要在 uClinux 上跑的应用都必须转换为 FLAT 格式。PC REDHAT LINUX 不支持这种格式的可执行文件，所以这些可执行文件都无法在 PC 上执行。

在 uClinux2.0.38 版本中，user 下的每个应用程序的 Makefile 的最后都有一句对$(CONVERT)宏的调用， 它用 elf2flt 工具将编译生成的 ELF 格式可执行文件转换为 uClinux 所支持的 FLAT 文件格式。而在目前所使用的 uClinux2.4 版本中，我们无法显式的看到这个转换的过程，其实它已经作为 arm-elf-gcc 的一个 LDFLAGS 了，而不必再显式调用了。这可在：**uClinux/vendors/ Winbond/W90N740/config.arch** 文件中看到有如下行：

```
LDFLAGS    = $(CFLAGS) -Wl,-elf2flt
```

下面，再来看看总体工作中 romfs 部分的工作，它实际就是在 user 下每个要参与编译的应用程序目录下执行：make romfs。

【注意】

这里首先第一个参与 make romfs 的目录就是 uClinux/vendors/ Samsung/740/，它的主要工作就是重新构建 romfs 下的内容，例如 romfs/dev/下的每个设备文件。具体参见下面的 uClinux/vendors/ Samsung/740/Makefile 中的 romfs 部分。

UClinux/user/下每个应用程序目录下的 Makefile 中都有如下：

```
romfs:
    $(ROMFSINST) /bin/$(EXEC)
```

注意到前面有宏定义：

ROMFSINST= romfs-inst.sh

romfs-inst.sh 是在 uClinux/tools 目录下提供的一个 shell 工具，它的作用就是将要烧写到板子上的内容复制到 uClinux/romfs/目录下，而这个目录下的

内容就是我们能够在板子上 ls 所能看到的所有内容。

下面再看看 image 部分的工作。这就是编译过程最后的一部分工作了，最终目的就是生成 image.bin 以供烧写或下载。
根据前面介绍，这部分工作就是在 uClinux/Winbond/W90N740 目录下执行 make image。因此下面看看这个目录下的 Makefile。

```
ROMFSIMG = $(IMAGEDIR)/romfs.img
IMAGE    = $(IMAGEDIR)/image.bin
ELFIMAGE = $(IMAGEDIR)/image.elf

DIRS =

ROMFS_DIRS = bin dev etc etc/config etc/default home lib mnt proc
usr var \
             home/httpd home/httpd/cgi-bin

DEVICES = \
    tty,c,5,0    console,c,5,1    cua0,c,5,64 cua1,c,5,65 \
    mem,c,1,1    kmem,c,1,2   null,c,1,3   zero,c,1,5   \
    random,c,1,8urandom,c,1,9                         \
    ram0,b,1,0   ram1,b,1,1                       \
    ptyp0,c,2,0 ptyp1,c,2,1 ptyp2,c,2,2 ptyp3,c,2,3 \
    ptyp4,c,2,4 ptyp5,c,2,5 ptyp6,c,2,6 ptyp7,c,2,7 \
    ptyp8,c,2,8 ptyp9,c,2,9 ptypa,c,2,10ptypb,c,2,11\
    ptypc,c,2,12ptypd,c,2,13ptype,c,2,14ptypf,c,2,15\
    ttyp0,c,3,0 ttyp1,c,3,1 ttyp2,c,3,2 ttyp3,c,3,3 \
    ttyp4,c,3,4 ttyp5,c,3,5 ttyp6,c,3,6 ttyp7,c,3,7 \
    ttyp8,c,3,8 ttyp9,c,3,9 ttypa,c,3,10ttypb,c,3,11\
```

```
    ttypc,c,3,12ttypd,c,3,13ttype,c,3,14ttypf,c,3,15\
    tty0,c,4,0  tty1,c,4,1  tty2,c,4,2  tty3,c,4,3  \
    ttyS0,c,4,64ttyS1,c,4,65                \
    rom0,b,31,0 rom1,b,31,1 rom2,b,31,2 rom3,b,31,3 \
    rom4,b,31,4 rom5,b,31,5 rom6,b,31,6 rom7,b,31,7 \
    rom8,b,31,8 rom9,b,31,9                 \
    ipsec,c,36,10                           \
    qspi0,c,126,0   qspi1,c,126,1   qspi2,c,126,2
    qspi3,c,126,3   \
    qspi4,c,126,4   qspi5,c,126,5   qspi6,c,126,6
    qspi7,c,126,7   \
    qspi8,c,126,8   qspi9,c,126,9   qspi10,c,126,10
    qspi11,c,126,11 \
    qspi12,c,126,12 qspi13,c,126,13 qspi14,c,126,14

all:
    dirs=$(DIRS) ; \
    for i in $$dirs ; do  make -C $$i || exit $? ; done
#这里的 all 其实什么也不作，因为我们不会在这个目录下执行 make。

clean:
    -dirs=$(DIRS) ; \
    for i in $$dirs; do [ ! -d $$i ] || make -C $$i clean; done

romfs:  #前面 user 下 make romfs 会执行这里的操作
    [ -d $(ROMFSDIR)/$$i ] || mkdir -p $(ROMFSDIR)
    for i in $(ROMFS_DIRS); do \
        [ -d $(ROMFSDIR)/$$i ] || mkdir -p $(ROMFSDIR)/$$i; \
```

```
    done
    for i in $(DEVICES); do \
        touch $(ROMFSDIR)/dev/@$$i; \
    done
    $(ROMFSINST) -s /var/tmp /tmp
    #$(ROMFSINST) ../../Generic/romfs /
    #$(ROMFSINST) ../../Generic/httpd /home/httpd
    #$(ROMFSINST) ../../Generic/big/inittab /etc/inittab
    #$(ROMFSINST) ../../Generic/big/rc /etc/rc
    echo "$(VERSIONSTR) -- " `date` > $(ROMFSDIR)/etc/version

image:
    [ -d $(IMAGEDIR) ] || mkdir -p $(IMAGEDIR)
    /usr/local/bin/genromfs -v -V "ROMdisk" -f $(ROMFSIMG) -d $(ROMFSDIR)
    $(MAKEARCH) -C $(ROOTDIR)/$(LINUXDIR) zImage
    cp $(ROOTDIR)/$(LINUXDIR)/arch/armnommu/boot/zImage $(IMAGEDIR)/
```

默认烧制的嵌入式 LiNUX 系统提供支持丰富的命令，如 :ps, vi，cat，reboot，mount/umount 等，并支持 gdbserver 调试手段。系统支持串口终端（minicom）。可用 cat 和 vi 命令来查看板子上的文件。

HHARM740 板上采用的是 ROMFS 的文件系统，它是只读的。可以扩展支持 JFFS 文件系统，它可读可写。华恒提供完备的 JFFS 支持软件包。

2.1.4 烧写 FLASH

烧写 flash 的方法有两种，一种是利用板子带的 JTAG 接口，使用 AJFlash 工具烧写；另一种是当 Bootloader 烧写进去以后，利用 Bootloader 的 flash 和 reload 命令对板上的 Flash 空间进行读写操作。

JTAG 工具

首次烧写或者 flash 上 bootloader 不可用时可使用 jtag 工具烧写。首先拔下电源，接上 JTAG 烧写器，并将串口连接到计算机的 COM1(/dev/ttyS0)，做好烧写准备。然后将光盘/bios/bootloader 复制到 jtag-rw 目录下,在 jtag-rw 目录下运行以下命令进行烧写:

```
cd /HHARM740-R1/jtag-rw
chmod 777 AJFlash
./AJFlash W
```

烧写完后拔掉电源，再拔去 jtag 烧写器接口。

为测试 Bootloader 烧写的效果， 启动 minicom(注意检查 minicom 是否设置波特率为 115200), 重启开发板，显示在 minicom 中的信息应如下所示：

```
Boot Loader Configuration:

        TFTP server port        : MAC 1
        Network phy chip        : DAVICOM DM9161E
        MAC 0 Address           : 00:00:00:00:00:01
        IP 0 Address            : 0.0.0.0
        MAC 1 Address           : 00:00:00:00:00:02
        IP 1 Address            : 192.168.2.24
        DHCP Client             : Disabled
        CACHE                   : Disabled
```

BL buffer base : 0x00300000
BL buffer size : 0x00100000
Press ESC to enter debug mode ......

Bootloader 烧写内核和文件系统

JTAG 方式烧写的速度比较慢，当将 bootloader 烧进 Flash 以后，可以用 flash 和 reload 命令方便的操作 Flash 文件的读写。

注意：

烧写时要将底板上靠外边的以太网口接上网线，使之能与 LINUX PC 相通，例如都接到一个 HUB 上或者直接用交叉线对接。

1. 配置:

1).set -net_mac 1

2).set -phy 0

3).set -ip1 192.168.2.24

4).set -dhcp 0

2. 下载内核

1).ft 7 linux.zip 0x7f020000 0x00008000 -acxz

2).换到 PC LINUX 的另外一个 TTY 口，进入 /HHARM740-R1/image/ 目录，或者是 /HHARM740-R1/uClinux/images 目录，执行如下命令：

#tftp 192.168.2.24
tftp>bin
tftp>put linux.zip

3. 下载根文件系统

1).ft 6 romfs.img 0x7f100000 0x7f100000 -a

2). 换到 PC LINUX 的另外一个 TTY 口，进入 /HHARM740-R1/image/ 目录，或者是 /HHARM740-R1/uClinux/images 目录，执行如下命令：

#tftp 192.168.2.24

tftp>bin
tftp>put romfs.img

2.1.5 板上嵌入式 LINUX 系统

板子 RESET 复位或者重新加电，系统启动信息如下：

```
Boot Loader Configuration:

        TFTP server port      : MAC 1
        Network phy chip      : DAVICOM DM9161E
        MAC 0 Address         : 00:00:00:00:00:01
        IP 0 Address          : 0.0.0.0
        MAC 1 Address         : 00:00:00:00:00:02
        IP 1 Address          : 192.168.2.24
        DHCP Client           : Disabled
        CACHE                 : Disabled
        BL buffer base        : 0x00300000
        BL buffer size        : 0x00100000
Press ESC to enter debug mode ......
Cache disabed!
Processing image 1 ...
Processing image 2 ...
Processing image 3 ...
Processing image 4 ...
Processing image 5 ...
Processing image 6 ...
```

Processing image 7 ...
Unzip image 7 ...
Executing image 7 ...
Linux version 2.4.20-uc0 (root@uClinux) (gcc version 2.95.3 20010315 (release)(4Processor: Winbond W90N740 revision 1
Architecture: W90N740
On node 0 totalpages: 4096
zone(0): 0 pages.
zone(1): 4096 pages.
zone(2): 0 pages.
Kernel command line: root=/dev/rom0
Calibrating delay loop... 39.83 BogoMIPS
Memory: 7MB = 7MB total
Memory: 5636KB available (1058K code, 205K data, 44K init)
Dentry cache hash table entries: 1024 (order: 1, 8192 bytes)
Inode cache hash table entries: 512 (order: 0, 4096 bytes)
Mount-cache hash table entries: 512 (order: 0, 4096 bytes)
Buffer-cache hash table entries: 1024 (order: 0, 4096 bytes)
Page-cache hash table entries: 2048 (order: 1, 8192 bytes)
POSIX conformance testing by UNIFIX
Linux NET4.0 for Linux 2.4
Based upon Swansea University Computer Society NET3.039
Initializing RT netlink socket
Starting kswapd
devfs: v1.12c (20020818) Richard Gooch (rgooch@atnf.csiro.au)
devfs: boot_options: 0x0
Winbond W90N740 Serial driver version 0.9 (2001-12-27) with no serial options edttyS00 at 0xfff80000 (irq = 6) is a W90N740

```
Blkmem copyright 1998,1999 D. Jeff Dionne
Blkmem copyright 1998 Kenneth Albanowski
Blkmem 1 disk images:
0: 7F100000-7F19E3FF [VIRTUAL 7F100000-7F19E3FF] (RO)
RAMDISK driver initialized: 16 RAM disks of 1024K size 1024 blocksize
loop: loaded (max 8 devices)
The flash size:0x00200000
Boot Loader Configuration:

        TFTP server port        : MAC 1
        Network phy chip        : PHY
        MAC 0 Address           : 00:00:00:00:00:01
        IP 0 Address            : 0.0.0.0
        MAC 1 Address           : 00:00:00:00:00:02
        IP 1 Address            : 192.168.2.24
        DHCP Client             : Disabled
        CACHE                   : Disabled
01 eth0 initial ok!
which:0
01 eth1 initial ok!
which:1
PPP generic driver version 2.4.2
PPP Deflate Compression module registered
PPP BSD Compression module registered
Linux video capture interface: v1.00
AM29LV160DB Flash Detected
usb.c: registered new driver hub
add a static ohci host controller device
```

```
: USB OHCI at membase 0xfff05000, IRQ 9
hc_alloc_ohci
usb-ohci.c: AMD756 erratum 4 workaround
hc_reset
usb.c: new USB bus registered, assigned bus number 1
hub.c: USB hub found
hub.c: 2 ports detected
usb.c: registered new driver W99683
W99683.c: v1.00 for Linux 2.4 : W99683 USB Camera Driver
NET4: Linux TCP/IP 1.0 for NET4.0
IP Protocols: ICMP, UDP, TCP
IP: routing cache hash table of 512 buckets, 4Kbytes
TCP: Hash tables configured (established 512 bind 512)
VFS: Mounted root (romfs filesystem) readonly.
Freeing init memory: 44K
blocksize:1024
blocksize:1024
blocksize:1024
blocksize:1024
Welcome to
          ____ _  _
         /  __| ||_|
   _   _| |  | |_ ____  _   _  _  _
  | | | | |  | || |  _ \| | | |\ \/ /
  | |_| | |__| || | | | | |_| |/    \
  |  ___\____|_||_|_| |_|\____|\_/\_/
  | |
  |_|
----------------------------------------------------------------------------------------
HHtech : An Embedded Linux Tech. Provider in Mainland China
```

```
For further information check:
http://www.uclinux.org/

MiiStationWrite 1
MiiStationWrite 1

Wait for auto-negotiation complete...OK
100MB - Full Duplex
MiiStationWrite 1
MiiStationWrite 1

Wait for auto-negotiation complete...OK
100MB - Full Duplex
#
# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:00:00:00:00:01
          inet addr:192.168.1.111  Bcast:192.168.1.255
               Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500
               Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 iB)  TX bytes:0 (0.0 iB)
          Interrupt:13

eth1      Link encap:Ethernet  HWaddr 00:00:00:00:00:02
```

```
          inet addr:192.168.2.111  Bcast:192.168.2.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:176 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:16343 (15.9 kiB)  TX bytes:0 (0.0 iB)
          Interrupt:14

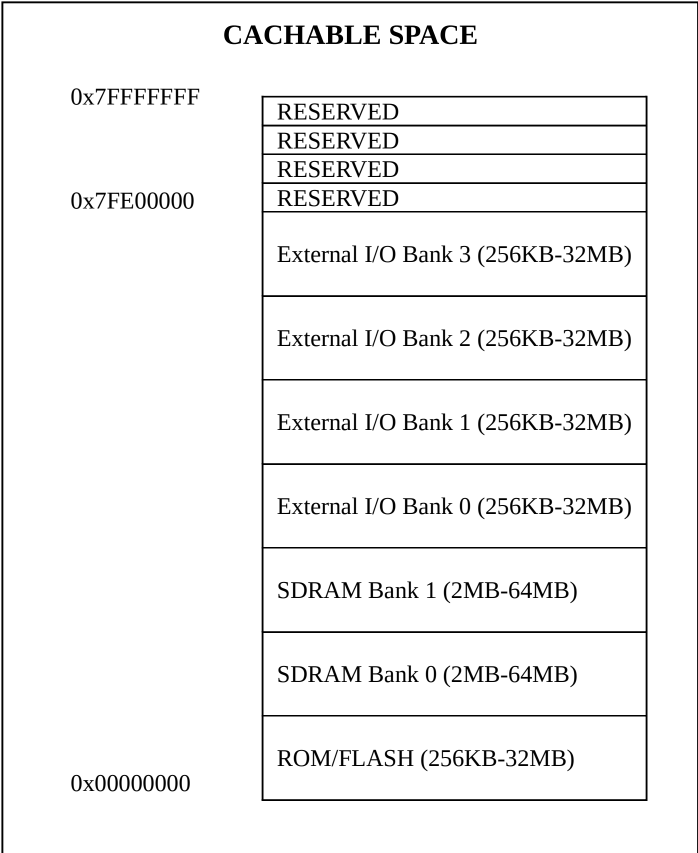
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 iB)  TX bytes:0 (0.0 iB)
#
```

注意：

eth0 对应的以太网口是底板上靠近串口的那个，即里面那个。

eth1 对应的以太网口是底板上靠外边的那个。

板上地址空间分布 Memory Map


CACHABLE SPACE
0x7FFFFFFF
RESERVED
RESERVED
RESERVED
0x7FE00000
RESERVED
External I/O Bank 3 (256KB-32MB)
External I/O Bank 2 (256KB-32MB)
External I/O Bank 1 (256KB-32MB)
External I/O Bank 0 (256KB-32MB)
SDRAM Bank 1 (2MB-64MB)
SDRAM Bank 0 (2MB-64MB)
ROM/FLASH (256KB-32MB)
0x00000000

NON-CACHABLE SPACE

Address	Region
0xFFFFFFFF	On-Chip APB Peripherals
0XFFF80000	On-Chip APB Peripherals
0xFFF00000	RESERVED
	On-Chip RAM (2KB,8KB)
0xFFE00000	External I/O Bank 3 (256KB-32MB)
	External I/O Bank 2 (256KB-32MB)
	External I/O Bank 1 (256KB-32MB)
	External I/O Bank 0 (256KB-32MB)
	SDRAM Bank 1 (2MB-64MB)
	SDRAM Bank 0 (2MB-64MB)
	ROM/FLASH (256KB-32MB)
0x80000000	

如上图所示。W90N740 的可寻址空间为 4GB，其中前 2GB 为可缓冲空间，后 2GB 为不可缓冲空间；片上的外围控制器占用最高的 1MB 空间，即 0XFFF00000-0XFFFFFFFF，片上的 RAM 从地址 0XFFE00000 开始，其

余片选可以被放在任意位置。(可缓存空间的 0X0-0X7FDFFFFF;非可缓存空间的 0X80000000-0XFFDFFFFF)。每个片选地址和大小均能通过修改相应寄存器的值而改变，但之间不能重叠。

本平台中，FLASH 将起始地址设在了可缓存空间的 0X7F000000 的位置；SDRAM 使用了 SDRAM BANK0，并将起始地址设为 0x0；其余片选由其它外部扩展功能占用。通过阅读内核代码，我们可以清楚地看到这一点。

下图反映了系统在内存中的映射关系。由图中我们可以看到，bootloader（bootloader)、压缩内核(linux.zip)，和根文件系统（romfs.img)，分成三个独立的部分烧写到 flash 中。bootloader 烧写的起始地址从 0x7F000000 开始，这样系统上电后自动由此处开始执行代码；压缩内核 linux.zip 被烧写到 07F0200000，在 bootloader 运行起来后将它解压复制到 SDRAM 的 0x00008000 的位置，执行内核启动，首先进行初始化工作；内核进行完初始化后把根文件目录 romfs 从 flash 中 mount 到 SDRAM 中，romfs 在 flash 中的位置为 0x7F100000。

2.1.6 串口支持

对有两个串口的开发版，可以利用第二个串口进行串口通信。

串口通信程序中需要对 termios 结构进行正确设置后方可进行通信;

在 user 目录下有 ttytest、serialtest 测试程序，可用于演示或在其基础上修改使用。

注意：第一个串口/dev/ttyS0 已被系统重定向为标准输入输出，因此不能做串口通信用。只能使用第二个串口，即/dev/ttyS1

2.1.7 USB HOST 支持

目前 USB HOST 接口提供支持的 USB 设备为 U 盘。要支持其它的 USB 设备，例如摄像头等设备，必须用户自己移植相应的驱动。

U 盘的支持需要注意的是必须选择 SCSI 支持。

```
SCSI support  --->
  [*] SCSI support
  --- SCSI support type (disk, tape, CD-ROM)
      [*]   SCSI disk support
     (40) Maximum number of SCSI disks that can be loaded as modules
      [ ]   SCSI tape support
      [ ]   SCSI OnStream SC-x0 tape support
      [ ]   SCSI CD-ROM support
   [*]   SCSI generic support
    --- Some SCSI devices (e.g. CD jukebox) support multiple LUNs
   [ ]   Enable extra checks in new queueing code
   [ ]   Probe all LUNs on each SCSI device
   [*]   Verbose SCSI error reporting (kernel size +=12K)
   [ ]   SCSI logging facility
  SCSI low-level drivers  --->
```

再有，就是 USB support --->

```
   [*] Support for USB
   [ ]   USB verbose debug messages
   --- Miscellaneous USB options
   [*]   Preliminary USB device filesystem
   [ ]   Enforce USB bandwidth allocation (EXPERIMENTAL)
   [ ]    Long timeout for slow-responding devices (some MGE
```

```
Ellipse UPSes)
   --- USB Host Controller Drivers
   [ ]   EHCI HCD (USB 2.0) support (EXPERIMENTAL)
   [ ]   UHCI (Intel PIIX4, VIA, ...) support
   [ ]   UHCI Alternate Driver (JE) support
   [ ]   OHCI (Compaq, iMacs, OPTi, SiS, ALi, ...) support
   [*]   SL811HS support
   [ ]   Try automatic configuration for SL811HS
   --- USB Device Class drivers
   [ ]   USB Bluetooth support (EXPERIMENTAL)
   [ ]   USB MIDI support
   [*]   USB Mass Storage support
   [*]     USB Mass Storage verbose debug
   [ ]     Datafab MDCFE-B Compact Flash Reader support
    [ ]     Freecom USB/ATAPI Bridge support
    ……
```

涉及的驱动代码：

uClinux-dist/linux-2.4.x/drivers/usb 目录下代码：

```
[root@localhost usb]# ls *.o
devices.o  drivers.o  hc_sl811.o  inode.o    usb-debug.o  usb.o
devio.o    hcd.o      hub.o       usbcore.o  usbdrv.o
```

uClinux-dist/linux-2.4.x/drivers/scsi 目录下代码：

```
[root@localhost scsi]# ls *.o
constants.o     scsi_dma.o          scsi_ioctl.o     scsi_mod.o
scsi_proc.o   scsi_syms.o  sg.o
hosts.o             scsidrv.o           scsi_lib.o         scsi.o
scsi_queue.o  sd_mod.o
scsicam.o        scsi_error.o    scsi_merge.o    scsi_obsolete.o
```

scsi_scan.o   sd.o

【注意】

这种 ls *.o 是常用的判断所用驱动代码的方法，那个代码被编译到内核了，就表明它被用到了。再从字面上就可以初步判断驱动是哪个或哪些文件了。驱动代码中涉及的修改为如下几行：

板子上插上 U 盘后，可通过 mount 命令加载 U 上的文件系统：

```
mount /dev/sda1` /mnt
```

2.2 软件应用开发

2.2.1 开发模式

在进行开发前，有必要先阐述一下宿主机和目标板的概念。宿主机是一台运行 LINUX 的 PC 机，目标板即华恒 HHARM740 开发板。

应用程序的开发有两种模式：

1）先在宿主机（Intel CPU）上调试通过后，再移植到目标板（740）上。这种 HOST 方式下有 gdb 调试工具可用，这对大型复杂应用是必不可少的。

移植的工作包括两个方面：

函数库的问题。由于开发平台提供的 libc 库同标准的 libc 库有一些不同，在程序移植时可能会有函数未定义的问题。对于这种问题，一般要求开发者自己编制这些要用到却又未定义的函数。

改动 Makefile 以适应目标板。

2）直接在目标板上进行开发（**通用开发模式，建议采用该模式**）。并将宿主机和目标板通过串口（建议 COM1）相连，在宿主 PC 机上运行 minicom 作为目标板的显示终端，mount 上宿主机硬盘，直接在目标板上调试应用。下面给出这种直接 TARGET 开发模式下的开发流程：

首先介绍一下嵌入式 LINUX 环境下应用程序开发的流程，以最简单

的串口打印字符串的 hello 程序为例：

（1） 用串口和网线（可不插 BDM 线）将宿主机和目标板连接起来。

（2） 在宿主机上 uClinux/user/下创建应用程序目录，例如就叫 hello；然后编辑 user 下的 Makefile，将 hello 这个应用程序的目录加入编译链表；下一步就是到 hello 目录下，编辑自己的 C 代码文件 hello.c，并在 hello 目录下的 Makefile 中加入如下一句：

```
cp –f hello /
```

返回 uClinux 目录，执行 make 进行编译，这样编译通过后的可执行程序 hello 就自动的被复制到宿主机/目录下以方便下面 mount 操作，避免 mount 时要进入很深的目录。

（3） 在宿主机上启动 minicom 作为目标板的仿真终端

（4）mount 映射宿主机硬盘根目录/到板子的/mnt 目录下,例如：mount 192.168.2.32:/ /mnt

（5） 在 minicom 下执行：

```
cd /mnt
./hello
```

调试信息通过串口打印在宿主机的 minicom 屏幕上或记录在 syslog 文件中，这样便可进行应用程序的调试。有问题，便切换去编辑、编译，只要不重启板子端就不必作任何操作，因为 mount 的宿主机硬盘上的应用程序会自动覆盖更新，再重新执行的就是更改后的新版本。这样反复调试、更改、编译再调试，而不必烧写板子，直至程序工作正常。

（6） 调试通过后，用 gdbtftpflash 下的./flash 将最终定版的 image.bin 烧写到板子上，则应用程序就会出现在板子的/bin 目录下。

【注意】

用户自己开发应用程序一定要放在 uClinux/user 目录下，否则，若放在其它目录下将有许多宏要自己定义，很是繁琐。

常用的工作方法：

grep string * -r

find –name filename

下面接着就上述流程中 hello 的例子具体介绍一下创建在 uClinux 软件系统中添加并编译自己的应用程序：

首先在 user 目录下创建名为 hello 的目录：

```
cd uClinux/user
mkdir hello
```

然后复制其它应用程序目录下的一个 Makefile 过来进行修改，例如：

```
cp ../get/Makefile .
```

修改这个 Makefile，将其中的 get 改为 hello 即可，其中可执行文件名可任意指定，不一定要为 hello，修改后的 Makefile 内容如下：

```
EXEC = hello
OBJS = hello.o

all: $(EXEC)

$(EXEC): $(OBJS)
	$(LD) $(LDFLAGS) -o $@.elf $(OBJS) $(LDLIBS)
	$(CONVERT)
	cp $(EXEC) ../../romfs/bin

clean:
	-rm -f $(EXEC) *.elf *.gdb *.o
```

【注意】

Makefile 中每行的缩进不允许用多个空格，而必须用 TAB 键，否则编译就会有莫名其妙的错误。

应用程序默认的栈的大小为 8K。可以在 Makefile 为其指定堆栈大小，例

如要增大该应用程序的堆栈容量，就在其 Makefile 中增加如下一句
FLTFLAGS = -s 65536
然后，编辑 hello.c 文件：

```
vim    hello.c
```

【注】上一步是用于生成新文件 hello.c。

```
#include <stdio.h>
int main()
{
        printf("Hello,guys!\n");
        return 0;
}
```

最后，修改 user 目录下的 Makefile，通知编译系统要编译用户新添加的这个应用程序目录 hello，即加入如下一行：

```
DIRSy+=hello
```

这样返回 uClinux 目录下，执行 make，则用户的应用程序就被编译进去了。
【注意】
一定要学会看 make 打印的错误。一般的，在有许多错误的情况下，一定要查看第一个 error，warning 是不需要理会的。

2.2.2 应用程序调试方法

直接在目标板上调试应用程序有两种办法：

打印串口：

这是嵌入式系统中最常用的调试手段，虽然简单但却有效实用。其实几种方法相比之下，最有效便捷的方法还是 printf，因为 gdb 远程调试一旦更改代码后还要烧写 FLASH（mount 上用 gdbserver 应该也可以），而且 gdb 调试的变量观察及设置断点都要手工输入命令，有这时间在 uClinux 下编译早已完成了。

使用 log 记录文件。例如使用 syslog 将应用程序运行过程中的中间信息全部记录在/var/log/syslog 下。

gdb 调试

板子上移植了 gdbserver，所以支持通过以太网或串口远程调试。下面介绍以太网调试步骤：（当然首先要求板子与宿主机的以太网可以互相 ping 通）

先在板子端启动 gdbserver

cd /bin

gdbserver 192.168.2.111:4444 myapp

【注】这里 myapp 为用户应用程序可执行文件名，设已烧制在板子/bin 目录下，4444 为 TCP 端口号，以后宿主机就通过这个端口与板子建立调试通道。（端口号可任取，一般 >2000）

然后在宿主机侧：

cd /HHARM740/uClinux/user/myapp

/HHARM740/uClinux/tools/arm-elf-gdb myapp.gdb

则进入 gdb 提示符：

(gdb) target remote 192.168.2.111:4444

上一步就是与板子建立 TCP 连接以进行调试，192.168.2.111 为板子出厂默

认的 IP。下一步进行调试时不要运行 run，而应该用 continue，即 c。

下面是调试板子上的 ifconfig 命令时板子上的信息记录：

```
/bin> gdbserver 192.168.2.175:4000 ifconfig
Process ifconfig created; pid = 68
code at 1a0058, data at 1a567c
Remote debugging using 192.168.2.175:4000
```

同时宿主 LINUX PC 机上的信息记录为：（PC 机上 ifconfig 的代码在 uClinux/user/route 下）

```
[root@localhost route]# /HHARM740/uClinux/tools/arm-elf-gdb ifconfig.gdb
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-linux-gnu --target=arm-bdm-elf"...
(gdb) target remote 192.168.2.175:4000
Remote debugging using 192.168.2.175:4000
0x1a0058 in _stext ()
(gdb) c
Continuing.

Program exited normally.
(gdb)
```

【注意】

要实现 gdb（或 ddd）的源代码级调试，必须在编译应用程序时打开-g 参数。这个要修改 uClinux/vendors/Samsung/740/1config.arch 文件来实现：

CFLAGS += –g

修改完毕后，在 uClinux 下执行 make，重编后，FLAT 格式的可执行文件大小没有变化，只是.gdb 文件变大了很多。这样在 gdb 和 ddd 环境中就可以看到原代码了。

或者是：

修改 uClinux/linux/.config 文件，使之有如下内容：

```
CONFIG_FULLDEBUG=y
```

同时 uClinux/linux/include/linux/autoconf.h 文件中有如下内容：

```
#define    CONFIG_FULLDEBUG     1
```

这样做和上面的方法是一个道理，都是为了加入-g 参数。

2.2.3 如何创建编译自己的应用

代码编写前应多阅读一下 user 目录下的类似应用程序的代码或从网上查找相关代码下载后阅读。

函数调用可参见《UNIX 环境高级编程》（见附录 B），因为书中所述为 UNIX 环境开发，但基本类似，具体应用到 LINUX 时请使用 man 查看帮助。uClinux 的 lib（uClinux/lib）虽然实现了很大部分的 C 函数，不过与标准的 linux 库还是有很大的区别的。而且由于实现的问题，可能会碰到一些莫名其妙的问题，这些可能是由于 uClinux 本身的库不完备造成的。在程序移植时可能会有函数未定义的问题。对于这种问题，一般要求开发者自己编制这些要用到却又未定义的函数。

uClinux 下所有的应用程序都采用静态连接。

Makefile 可以从 user 下某个目录中的 Makefile 作为样本，稍做改动即可使用。

修改 user 目录下的 Makefile，使其将用户的应用程序目录加入编译队列。具体方法：加入如下行：DIRSy += dir

其中 dir 是你的应用程序源码目录。注意，要在你自己的应用程序的 Makefile 文件中加入拷贝命令以拷贝可执行文件到 romfs/bin 目录下。

下面举例说明：

在 user 目录下创建 hello 的目录

```
cd /HHARM740/ uClinux/user
mkdir hello
```

修改 user 下 Makefile，加入：

```
dir_y += hello
```

编写 hello.c，只是打印串口 hello 字符串。Makefile 则以 user/ping 的 Makefile 为模板，将两处 ping 改为 hello 即可，Makefile 见下：

```
EXEC = hello
OBJS = hello.o
LTFLAGS = -s 8192
all: $(EXEC)
$(EXEC): $(OBJS)
	$(CC) $(LDFLAGS) -o $@ $(OBJS) $(LDLIBS)
romfs:
	$(ROMFSINST) /bin/$(EXEC)
clean:
	-rm -f $(EXEC) *.elf *.gdb *.o *~
```

然后退到/HHARM740/uClinux/目录下执行 make，再用 gdbtftpflash 工具将编译生成的/tftpboot/image.bin 烧写到 FLASH 中重启，则板子的/bin 目录下就会看到 hello 可执行应用。

【注意】

用户自己开发应用程序一定要放在 uClinux/user 目录下，否则，若放在其

它目录下将有太多的宏要自己定义，非常繁琐。而且编译时一定要返回 uClinux 进行编译，而不能直接在自己应用的目录下编译。

Makefile 中的缩进一律要用 Tab 键，不允许有任何空格键。

2.2.4 如何移植软件

在 LINUX 这个开放的世界里，软件开发的工作量大量的表现在自由软件的移植上。华恒软件包中提供了许多从网上下载的应用程序，它们大部分并没有被编译，用户若需要使用其中的软件，就要将它们加入应用程序的编译链表。

uClinux/user/Makefile 决定某个应用程序是否被编译。它告知编译器哪个目录要加入编译列表，uClinux/romfs.mk 则决定将哪个编译生成的可执行文件复制到 uClinux/romfs/bin 中，并最终烧制到板子上，uClinux/vendors/Motorola/M5272 下的 vendor.config 文件定义了各个宏，例如要编译 sash，则：CONFIG_USER_SASH_SH = y。

下面以移植 microwin 为例说明：

在 uClinux 目录下执行 make menuconfig，选择更改应用程序配置，不需要更改内核配置。如图 5-9 至 5-13 所示。


uClinux v1.3.3 Configuration
Main Menu
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help.
Legend: [*] built-in [ ] excluded <M> module < > module capable
Target Platform Selection --->

Load an Alternate Configuration File
Save Configuration to an Alternate File
<Select> < Exit > < Help >

图 5-9 配置主选单


uClinux v1.3.3 Configuration
Target Platform Selection
Arrow keys navigate the menu. <Enter> selects submenus --->.
Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes,
<M> modularizes features. Press <Esc><Esc> to exit, <?> for Help.
Legend: [*] built-in [ ] excluded <M> module < > module capable
--- Choose a Vendor/Product combination.
(HHtech/M5272) Vendor/Product
(uClinux-2.4.x) Kernel Version
--- Library is uC-libc (old)
[ ] Default all settings (lose changes)
[ ] Customize Kernel Settings
[*] Customize Vendor/User Settings (NEW)
[*] Update Default Vendor Settings
<Select> < Exit > < Help >

图 5-10 配置选单



图 5-11 保存配置提示界面



图 5-12 应用程序配置主选单

选择 MircroWindows 后其中的选项配置如下所列：

[*] MicroWindows

--- Compiling Options

[*] Optimize

[ ] Debug

[ ] Verbose

--- Libraries

[*] Microwin

[*] NanoX

[ ] SharedLibs

[ ] NWidget

[ ] OBJFramework

--- Demos

[*] MicrowinDemo

[*] NanoXDemo

--- Applications

[*] NanoWM

--- Settings

(CONFIG_USER_MICROWIN_MWPF_TRUECOLOR0888) Screeen PixType

[ ] Link App into server

[ ] Have File IO

[*] Have BMP Support

[*] Have GIF Support

[ ] Have PNM Support

[ ] Have XPM Support

[ ] Have JPEG Support

[ ] Have PNG Support

[ ] Have T1LIB Support

[ ] Have FreeType Support

[ ] Have SharedMemory Support

[ ] Have Han Zi Ku font Support

[ ] Have Big5 Support

[ ] Have GB2312 Support

[ ] Have MS Fonts

[ ] Build Screen Driver only

[ ] Window Erase Move
[ ] Window UpdateRegions Move
[ ] Gray Palette
--- Display Config
[ ] X11 Display
[*] Frame Buffer Display
[ ] Frame Buffer VGA
[ ] VT Switch
[ ] Portrait Mode
[ ] Frame Buffer Reverse
[ ] VGA Lib
[ ] HW VGA
[ ] Cleopatra VGA
--- Mouse/Touch Screen
[ ] GPM Mouse
  [ ] Serial Mouse
  [ ] TP Mouse
  [ ] TP Helio
  [ ] ADS Mouse
  [ ] IPAQ Mouse
  [ ] Harrier Mouse
  [ ] PSION Mouse
  [ ] uClinux/Palm TS
  [ ] Cleopatra Mouse
  [*] No Mouse
--- Keyboard
  [ ] TTY Keyboard
  [ ] Scan Keyboard

[ ] Pipe Keyboard

[ ] Cleopatra Keyboard

[*] No Keyboard

--- Install These Applications

[ ] Landmine

[ ] Launcher

[ ] Move

[ ] MTerm

[ ] MTest

[ ] MTest2

[ ] MUserFd

[*] Nano-X

[*] NanoWM

[*] NPanel

[*] NTerm

[*] NTest

[ ] NTetris

[*] NXclock

[ ] NXev

[ ] NXkbd

[ ] NXlsclients

[ ] NXterm

[ ] NXView

[ ] Slider

[ ] VNC

[ ] World

然后退出，



图 5-13 退出界面

选择 Yes，退出，make 即可在 uClinux/romfs/bin/下看到：

-rwxr-xr-x　1 root　root　10079　9 月　2 11:55 **nanowm**

-rwxr-xr-x　1 root　root　51534　9 月　2 11:55 **nano-X**

在板子终端 minicom 下执行：nano-X&，然后执行 nanowm&，再执行 npanel 或 nxclcok 即可看到演示的窗口。其中 nano-X 为 X-server，nanowm 相当于一个 X-WINDOW 的客户端，如 KDE 等，而 npanel 为 X 上的应用程序。

上述的 microwin 是个特殊的例子，实际上大量的 LINUX 下的应用软件的移植都要涉及配置文件（.conf）文件的编制，例如 DHCP 服务器、PPTP VPN 服务器、PPP 拔入服务器、PPP 拨号程序、WEB SERVER（boa）等应用软件，都需要编写正确的配置文件，否则这些软件就无法正常运行。这些配置文件一般的应用软件目录下都在 example、config 之类的目录下提供了样板和例子文件，只需在这些例子文件基础上略做修改即可。

还有，就是这个配置文件放置的目录，配置文件必须放到正确的目录下才能为应用程序所正确的读取，否则还是没有用处。这些文件放置目录的定义一般的都在一些.h 文件里面给出，例如常见的有 pathname.h、define.h 等，有些特殊的则是以宏的形式在 Makefile 里给出。例如 user/pppd/pppd/ pathname.h ， user/dhcp-isc/includes/dhcpd.h ， user/init/ pathnames.h ， bash/pathnames.h，telnetd/pathnames.h 等。

2.2.5 关于多进程

由于 uClinux 没有 mmu 管理存储器，在实现多个进程时（fork 调用生成子进程）需要实现数据保护。uClinux 只支持 vfork。uClinux 的 fork 等于 vfork。实际上 uClinux 的多进程管理通过 vfork 来实现。这意味着 uClinux 系统 fork 调用完程后，要么子进程代替父进程执行（此时父进程已经 sleep）直到子进程调用 exit 退出，要么调用 exec 执行一个新的进程，这个时候将产生可执行文件的加载，即使这个进程只是父进程的拷贝，这个过程也不能避免。当子进程执行 exit 或 exec 后，子进程使用 wakeup 把父进程唤醒，父进程继续往下执行。

下面给出一个典型的用 vfork 实现的多进程代码：

```
int pid;
if ((pid=vfork())<0){
        printf("vfork error\n");
 }
 else if (pid==0){//子进程
        execl("/bin/ifconfig","ifconfig",(char*)0);
        exit(0);
 }
 else{//父进程
  //若需要等待子进程返回，则需要调用 waitpid
  if(waitpid(pid,NULL,0)!=pid)
  printf("wait pid error! ");
……
 }
```

uClinux 的这种多进程实现机制同它的内存管理紧密相关。uClinux 针对 nommu 处理器开发，所以被迫使用一种 flat 方式的内存管理模式，启动新

的应用程序时系统必须为应用程序分配存储空间，并立即把应用程序加载到内存。缺少了 MMU 的内存重映射机制，uClinux 必须在可执行文件加载阶段对可执行文件 reloc 处理，使得程序执行时能够直接使用物理内存。

2.3 设备驱动开发

华恒 ARM 嵌入式 LINUX 系列开发套件都提供接口底板的电路图，并有和核心板接口的 120 芯插座信号的详细定义，这样用户可以通过定制底板来扩展自己需要的接口，例如，可外接 ISA 总线的以太网控制芯片来增加以太网接口、扩展多串口、扩展 A/D 数据采集接口、扩展红外接口等等。所有这样硬件接口的扩展都需要在操作系统软件上提供设备驱动程序的支持，从而可以为上层软件所访问。

在 LINUX 下，所有的设备都被当做文件处理，即设备文件。设备驱动就是向 OS 注册这个设备文件的所有操作，因此其框架非常简单，主要就是两部分：

1 一个 xxx_init 函数，完成基本一些初始化，向操作系统注册这个设备文件所需要的操作，就是 file_operation 结构中填写的函数指针。这个 xxx_init 函数加到 mem.c 中，从而完成和操作系统的挂接。

2 剩下就是 file_operation 结构中注册的文件操作函数的实现代码了，这构成设备驱动.c 文件的绝大多数代码。主要的有 xxx_open,xxx_read,xxx_write。对于以太网就是 xxx_tx,xxx_rx。

字符设备驱动的典型例子可以参见 usb.c。

下面设备驱动开发的介绍均基于 uClinux 2.0 内核版本，2.4 内核的设备驱动与 2.0 基本相同，只是略有区别。

在具备了基本的概念之后，驱动开发最为快捷有效的方式就是在 uClinux/linux/drivers/下的各种驱动中找到与自己设备相类似的驱动代码，阅读代码框架结构，并以此为基础进行修改从而使之变为自己的驱动程序。

2.3.1 简介

Linux 的设备管理（即输入输出子系统）是操作系统的重要组成部分。与进程管理、内存管理、和文件系统相比，设备管理相对来说要"乱"一些。这主要是应为存在多种多样的输入输出设备，导致难以形成一个通用的解决方案。尽管如此，输入输出子系统的基本功能就是要提供一个统一而简单的输入输出系统调用接口。

输入输出子系统可分为上下两部分：一部分是下层的、设备相关的，即所谓的设备驱动程序，它直接与相应设备打交道，并向上提供一组访问接口。另一部分是上层的、设备无关的，这部分根据输入输出请求，通过特定设备驱动提供的接口，与设备进行通信。

虽然设备种类繁多，但是为了便于使用，输入输出子系统必须提供一个简单统一的设备使用接口。与其它 UNIX 一样，LINUX 将各种设备都作为特殊文件来处理。也就是说，对设备可以进行 read 和 write 等操作。这些都是由输入输出子系统的设备无关部分来完成的。

为 LINUX 内核编写驱动程序，并不像其他操作系统那么复杂。实际上，我们所要做的只是为相应的设备编写几个基本函数并向 VFS 注册即可。当上层应用要使用该设备时，VFS 就会调用相应的设备函数。

LINUX 下设备驱动程序的编制和 pSOS 下写 BSP 类似，一般的工作方式都是在一个现成的驱动程序的基础上针对特殊的硬件设备做相应的改动，

并不是要求从头编起，当然，还是要求对设备驱动程序有相当的了解。设备驱动多数是基于中断的（安装中断处理函数，即注册对应置中断向量表，使用 LINUX 提供的注册函数 request_irq）；但也有基于轮询方式的；有的甚至是在运行时动态切换，如并口驱动。

外设 I/O 空间编址

根据内存空间是否独立，可以将 I/O 空间的配置分为两种情况：一种是 I/O 空间与内存空间相互独立，这样 I/O 空间的访问需要使用专门的 I/O 语句如 inb 和 outb 等。INTEL CPU 就使用这种方法。另一种事将 I/O 寄存器作为内存的一部分，这样使用普通的内存访问语句就可以读写 I/O 寄存器。Motorola 680x0 就采用这种体系结构。我们华恒板子的 MC68EZ328 就是这样。即其 RAM，FLASH，外设 I/O 均参与统一平坦编址，也没有地址变换。就像 DOS 等。

模块支持

标准 PC 上的 LINUX 可以以模块的形式加载各种类型的设备。

要求在设备驱动中编制两个入口点：

```
init_module();
cleanup_module();
```

与内核相关的所有内容都在：

uClinux/linux/include/linux

uClinux/linux /include/asm

两个目录下的头文件中。

uClinux 2.0 内核并不支持模块化 lkm：（loadable kernel modules），即不支持模块化载入，因此只能将设备驱动（.o）静态的编入内核。这就要求修改内核驱动部分编译时的 Makefile，具体参见后面例子。由于 uClinux 只支持静态链接的设备驱动，它与内核编为一体，不需要注销（cleanup_module）。只有按模块（lkm）加载的设备，才有必要注销。

设备分类

设备驱动程序通常可归类为以下三种类型：

字符设备（char）：以字节为单位；只能按顺序访问；不用缓存。

块设备（block）：以块为单位；允许随机访问；多用缓存技术。

网络接口（net）

其它设备驱动程序模块，都在 uClinux/linux/drivers/目录下有分类的子目录，如：cdrom，scsi，pci，isdn，sbus，sound。对于 2.4 内核，则分类的子目录要更多。LINUX 的文件系统：实际上就是一种软件设备驱动程序（如 ROMFS 文件系统使用的 blkmem.c 和 JFFS 文件系统使用的 flash.c）。

所有的字符和块设备的驱动程序都支持文件操作的接口，因此用户对任何一个设备的存取都如同对文件操作一样，即把每一个设备都当作一个特殊文件对待。因为内核对设备的操作时通过文件接口的，因此只要特定设备的驱动程序支持这一抽象的文件接口即可。在内核中增加一种新的设备驱动程序是相对容易的。

内核空间与用户空间

设备驱动编程实际就是对 LINUX 内核编程，设备驱动都是以内核态在内核空间运行。若驱动代码编制不良，就会导致整个系统死掉，例如：

Kernel panic……

设备驱动程序接口

LINUX 把每一个设备都当作一个特殊文件对待，存放在/dev/目录下。输入输出子系统向内核其它部分提供了一个统一的标准设备接口。这是通过数据结构 file_operation（include/linux/fs.h）。

```
struct file_operations{
	int (*lseek)(struct inode *,struct file *,off_t,int);
	int (*read) (struct inode *,struct file *,char *,int);
	int (*write) (struct inode *,struct file *,const char *,int);
	int (*readdir) (struct inode *,struct file *,void *,filldir_t);
	int (*select) (struct inode *,struct file *,int,select_table);
	int (*iocrl)
	int (*mmap)
```

```
    int (*open)
    int (*release)
    int (*fsync)
    int (*fasync)
    int (*check_media_change)(kdev_t dev);
    int (*revalidate)(kdev_t dev);
};
```

2.3.2 设备驱动程序的框架

SVR4 曾提出 DDI/DKI 规范，用以规范化设备驱动程序与内核之间的接口。LINUX 的设备驱动与内核的接口与 DDI/DKI 规范类似，可分为三部分
与内核接口：通过 file_operations 来完成。
与系统启动代码的接口：对设备进行初始化。
与设备的接口。即对设备的读写等操作。

以 uClinux 为例，系统引导时，通过 sys_setup（linux/fs/filesystems.c）进行系统初始化。而 sys_setup 又调用 device_setup（linux/drivers/block/genhd.c）进行设备初始化。这里又可分为字符设备的初始化和块设备的初始化。字符设备初始化由 chr_dev_init（linux/drivers/char/mem.c）完成。块设备初始化由 blk_dev_init（linux/drivers/block/ll_rw_blk.c）完成。对于字符设备，chr_dev_init 中又分别对不同类别的字符设备进行初始化，如：tty_init，lp_init，misc_init 等，USB 驱动的初始化代码就在这里调用 usb_init。其中 misc_init（linux/drivers/char/misc.c）中又对多种属于 misc 类的设备（如各类鼠标）进行初始化。例如若加个手写板的驱动，手写板就可划为 misc 类字符设备，因此手写板的设备驱动代码中提供的初始化函数，如 handpad_init 函数就应该加到 linux/drivers/char/misc.c 文件中的 misc_init 函数中进行调用。而

设备的注册就在这些 init 函数中被调用，例如 misc 类字符设备的 init 函数中就要调用 misc_register，例如 busmouse.c 中的 bus_mouse_init 最后就调用 misc_register。而对于其他非 misc 类字符设备，就在其 init 函数中调用 register_chrdev。例如：打印驱动 lp.c 中的 lp_init 中就调用 register_chrdev。

2.3.3 设备驱动编程注意事项

库函数使用

由于内核是个自我封闭的系统，无法使用 libc 中提供的标准库函数，内核重新封装实现了一些常用的库函数，如字符串操作，memcpy 等，这都是经过多人高度优化的代码，可读性不是很强，但性能却是最佳，这也正是内核所追求的。

内存分配

内核态下编程也要使用栈，即内核栈，只有 8K 大小。且一旦栈溢出，就可能会导致内核崩溃。因此使用栈一定要小心。对于大数组使用，应该用静态变量或分配到堆上。

I/O 空间检查

在为设备分配要使用的 I/O 地址时，需要检查该地址空间是否已经被其它设备所占用。即：request_region。

睡眠唤醒队列 interruptible_sleep_on

在设备驱动代码中经常遇到要等待数据到达的情况，如 DMA 操作，但内核态不能陷入死循环等待，它必须要处于这样一种状态：等待但又不占用 CPU。

LINUX 支持这种工作模式，驱动程序处于睡眠状态并交出 CPU，这时该进程被放入一个睡眠等待唤醒队列。在中断处理程序中有数据到达时，就从队列中找到该进程并唤醒之。

2.4 内核和 2.0 内核代码中对睡眠唤醒队列的定义有所不同，但调用函数是相同的。(具体可阅读 uClinux/drivers/下面的具体的驱动代码，大多数驱动代码中都要使用睡眠唤醒队列)

在 2.0 内核下睡眠唤醒队列的定义和使用如下：

struct wait_queue *wq = NULL;

在 2.4 内核下，睡眠唤醒队列的定义和使用如下：

DECLARE_WAIT_QUEUE_HEAD(wq);

睡眠唤醒队列的使用和调用函数都相同，即：

interruptible_sleep_on(&wq);

wake_up_interruptible(&wq);

设备的中断

轮询方式要做在应用程序层，只有采用中断方式的设备驱动才需要在内核中完成。一般的，可以在打开设备时调用 **request_irq** 注册一个中断，在关闭设备时调用 **free_irq** 注销这个中断。

实现 file_operations 指定的接口操作函数。

初始化、注册等操作完成后，才开始真正构建设备驱动程序的大部分代码，即通常要实现的操作，如 open，read，write，ioctl 等。

2.3.4 添加自己的设备驱动

字符设备驱动的典型例子有：

串口驱动（uClinux/linux/drivers/char/mcfserial.c）、

打印驱动（uClinux/linux/drivers/char/lp.c）、

USB 设备驱动（uClinux/linux/drivers/char/usb.c）等。

块设备驱动的典型例子有：

ROMFS：uClinux/linux/drivers/block/blkmem.c

JFFS 驱动：uClinux/linux/drivers/block/flash.c

硬盘驱动：uClinux/linux/drivers/ide/ide.c，ide-disk.c 等。

以太网驱动的典型例子：

FEC 驱动：uClinux/linux/drivers/net/fec.c

添加一个自己的设备驱动的具体步骤：(仅适用于字符设备和块设备，网络设备除外)

在/dev/下用 mknod 手工创建一个设备，设备名即下面要注册使用的设备名。例如：

mknod usb c 23 0

其中 c 代表字符设备，23 代表主设备号，0 代表次设备号。

对于 2.4，则是要用 touch，例如：

touch @usb,c,23,0

uClinux 自带工具集中的 genromfs 工具会将@usb 自动转换为 usb。

usb 这个名字要与下面注册函数中使用的字符串统一。

在驱动代码（例如 usb.c）中编写该设备的初始化函数，一般为 xxx_init，例如 usb_init。将将这个函数加到 uClinux/linux/drivers/char/mem.c 文件的 chr_dev_init 函数中，chr_dev_init 函数将在 LINUX 操作系统启动过程中被调用，它完成所有字符设备驱动的初始化。同理块设备的初始化函数调用要加到 uClinux/linux/drivers/block/ll_rw_blk.c 文件的 blk_dev_init 函数中，这个函数也是在 LINUX 操作系统启动时调用执行，它完成所有块设备的初始化调用。同理，所有以太网设备驱动的初始化 xxx_probe 函数调用都要加到 uClinux/linux/drivers/net/Space.c 中去。在这些初始化函数中要完成如下工作：

1）填充 file_operations 数据结构，指定要实现向上提供的接口操作。并以此为参数，用 register_chrdev 和 register_blkdev 向操作系统注册。实际上就是向内核注册一个设备文件（主设备号/次设备号）及其一系列标准操作

接口。注册的工作就是向 LINUX 管理的设备链表中加入一个节点。
2）外设芯片的片选、中断等相关的初始化工作。
3）用 request_irq 向操作系统申请中断。
4）外设芯片自身所需要的一些硬件初始化工作。
剩余就是文件中（例如 usb.c）大量的编码工作，完成那些向操作系统注册的接口函数以及中断处理函数的编码。这些代码都是和外设芯片的硬件特性密切相关的。它们构成了设备驱动.c 文件的大部分代码量。
最后，以 usb.c 为例，要修改 uClinux/linux/drivers/char/Makefile，在 L_OBJS 中添加 usb.o,即将 USB 的设备驱动 usb.c 加入编译,并静态的链接到 LINUX 内核中去，添加后的如下示：
L_OBJS	:= tty_io.o n_tty.o tty_ioctl.o pty.o mem.o usb.o

2.4 配置文件保存的方法

华恒发行套件中的 uClinux 采用 ROMFS 作为其根文件系统，因此板上目录是不可写的。只有/var，/tmp 因为是 RAM 盘可写，但板子一掉电里面的内容就丢失了，因此只能作临时文件保存，无法永久的保存数据，例如系统配置文件等。下面大概介绍一下几种 FLASH 上保存配置文件的方法：

1. 对于简单的数据，小的配置文件等，而且不是非常频繁的（例如一分钟写 10 次）写入的，可以直接自己在 FLASH 的空余处（例如第二片 FLASH 上）划出一块区域，以自己定义的方式写入，并在板子启动时自动读出。华恒提供了这样的样例代码，即 user/memtools/，其详细介绍在手册第二章“ FLASH 扇区保存 IP 地址 ”一节。这种方式最大的好处是代码清晰简单，用户的控制程度最大，形式最灵活，保存的可以是自定义的数据，可以不是文件的形式。而其它的几种方式都是要求以文件的形式保

存，无法处理自定义的数据保存。

下面以在板子 FLASH 保存 IP 地址配置文件为例予以说明：

在 uClinux/user/memtools 这个目录下提供了设置板子 IP 的配置文件 /etc/config/start 的 FLASH 扇区保存和启动时恢复的两个工具软件：save 和 recover。

这个例子的工作原理如下：

板子上的/etc/config/start 配置文件是用来设置板子 IP 的 rc 文件，/etc/config 是 RAM 盘，可读可写。

板子的 IP 先后由两个文件设置它：先是/etc/rc，然后再执行/etc/config/start。这个顺序是由 uClinux/user/init/simpleinit.c 中的启动代码决定的。

当用户的应用程序（例如 WEB/CGI）修改了板子的 IP，并记录到 /etc/config/start 文件后，需要执行 save 命令，则将/etc/config/start 文件保存到板子第二片 FLASH 的一个扇区，即起始地址为：0xFFEF0000 的扇区。

数据保存的格式为：

文件名（32 字节）＋文件长度（4 字节）＋文件内容

每次板子重启时，都要执行/etc/rc 这个脚本文件，其中有一句执行

/bin/recover

这个程序根据上述数据保存的格式，从板子第二片 FLASH 的最后一个扇区中恢复出/etc/config/start 文件的内容，紧接着系统就执行/etc/config/start 中的内容，从而完成对板子 IP 的二次修正。

下面给出/etc/rc 和/etc/config/start 两个文件的内容：

```
/etc> cat rc
hostname HHCF5272-3ETH-R1
/bin/expand /etc/ramfs.img /dev/ram0
/bin/expand /etc/ramfs.img /dev/ram1
mount -t proc proc /proc
mount -t ext2 /dev/ram0 /var
mount -t ext2 /dev/ram1 /etc/config      【作为 RAM 盘】
```

```
mkdir /var/tmp
mkdir /var/log
cp /etc/default/* /etc/config
/bin/recover        【从第二片 FLASH 的一个扇区恢复/etc/config/start 文件】
ifconfig lo 127.0.0.1
ifconfig lo up
ifconfig eth0 192.168.2.111        【对板子 IP 的第一次设置】
textout
```

/etc/config> cat start

```
ifconfig eth0 192.168.2.151 netmask 255.255.255.0  【对板子 IP 的二次修正】
```

这样板子启动后的 IP 为 192.168.2.151，而不是第一次设定的 192.168.2.111。

软件的可扩展性：

若用户需要保存的数据或文件比较大，一个扇区（64K）无法容下，则可以在这个软件例子模板的基础上很轻松的扩展到多扇区，多文件保存。

下面是 save.c 的 main 函数代码：

```
int main(int argc,char *argv[]){
        unsigned long addr;
        addr = 0xffef0000;
        erase_sector(addr);
        sleep(3);      //这里是为了等待 FLASH 擦除后稳定下来变为 FF。
        write_file_to_flash("/etc/config/start",addr);
}
```

下面是 recover.c 的 main 函数代码：

```
int main(int argc,char *argv[]){
        unsigned long addr;
        addr = 0xffef0000;
        recover_file_from_flash(addr);
}
```

由上代码可见，若要增加扇区保存，唯一要改动的就是

addr = 0xffef0000;

且占用多个扇区就要多次调用 erase_sector(addr)分别擦除各个扇区。
而且由于定义的数据保存格式中含有全路径文件名和文件长度，因此很容易的扩展到多文件保存/恢复的应用场合。
代码中用到的两个基本 FLASH 操作功能函数 API：

1. 1）erase_sector(addr)　　【扇区擦除，addr 为扇区起始地址】

   2）write_word(addr,value)　　【向 addr 地址写入 value 的双字节数值】

   注意：addr 限制为第二片 FLASH 地址范围内，第一片 FLASH 为操作系统，不要破坏。

有了这两个基本功能单元，就可以实现 FLASH 上的所有操作。

2. 对于比较多的配置文件，一般的都先写在 RAM 盘中，然后选择保存，一次性写入 FLASH 的某几个扇区。这时就可使用 flatfsd 软件，它的使用需要内核的配合支持，即要在 blkmem.c 中为其指定保存数据的几个扇区的起始/结束地址。这种方式也不能适应非常频繁的写入。下面举例说明：板子上所有应用程序的配置文件都在/etc/config 目录下，它是个 RAM 盘，每次用户修改了这些配置文件后，都要执行一次保存的操作，而保存操作就是将/etc/config/目录下所有的配置文件烧写到板子第一片 FLASH 的最后四个扇区中，因为 LINUX OS 只占用了第一片 FLASH 的一部分，后面的多个扇区都没有用到，可以用来自定义使用。板子上另一个目录为/etc/default/，它是 ROMFS 文件系统，里面保存了所有应用程序的出厂配置。宿主 PC 机上 config 目录是空的，板子启动后自动加载 RAM 盘中的内容将这个目录覆盖。板子出厂时配置存储区（第一片 FLASH 最后 4 个扇区）为空，板子第一次启动时会执行一个从 default 到 config 的 COPY 操作（由 user/flatfsd/newfs.c 中完成）。但以后就不再执行 COPY 操作了，一律从配置存储的 4 个扇区中读取配置信息，并在 config 目录下生成对应的配置文件。例如启动时打印如下信息：

FLATFSD: created 33 configuration files (5816 bytes)

要采用这种 FLASH 数据保存方式，就必须修改 blkmem.c 文件，在其中开辟几个个扇区以保存配置文件，这几个扇区要可读可写，这时的 blkmem.c 中就提供了对这四个扇区的读写驱动：

blkmem.c 实际是 FLASH 的设备驱动。这几个扇区实际是个微型的 FLAT FS 文件系统。

下面的例子在板子第一片 FLASH 上开辟最后四个扇区，修改了 blkmem.c，只增加了加黑部分：

```
struct arena_t {
        int rw;
        unsigned long address; /* Address of memory arena */
        unsigned long length;   /* Length of memory arena. If -1, try to get size from romfs header */
        program_func_t program_func; /* Function to program in one go */
        xfer_func_t read_func; /* Function to transfer data to main memory, or zero if none needed */
        xfer_func_t write_func; /* Function to transfer data from main memory, zero if none needed */
        erase_func_t erase_func; /* Function to erase a block of memory to zeros, or 0 if N/A */
        unsigned long blksize; /* Size of block that can be erased at one time, or 0 if N/A */
        unsigned long unitsize;
        unsigned char erasevalue; /* Contents of sectors when erased */
        /*unsigned int auto_erase_bits;
        unsigned int did_erase_bits;*/
} arena[] = {
```

```
#ifdef INTERNAL_ROMARRAY
        {romarray, sizeof(romarray)},
#endif

#if defined(CONFIG_COLDFIRE) && !defined(CONFIG_ROMKERNEL)
/*
  *     The ROM file-system is RAM resident on the ColdFire eval boards.
  *     This arena is defined for access to it.
  */
      {0, 0, -1},
#define FIXUP_ARENAS     arena[0].address = (unsigned long) &_ebss;
……
```

//开辟第一块 FLASH 的最后四个扇区做数据保存之用：

```
#ifdef CONFIG_HHTECH
      {1,0xFFdc0000,0x40000,0,0,write_pair,erase_pair,0x10000,0x40000,0xff},
#endif
……
};
```

内核驱动部分就只需要增加上述三行，剩下就是应用层代码 flatfsd 的工作了。然后，在应用层通过 kill flatfsd 进程来实现保存到 FLASH 中的工作。

system1("/bin/killall -10 flatfsd");

它退出时调用 flatwrite

Write out the contents of the local directory to flat file-system.

在/etc/rc 中加入如下一句：

```
/bin/flatfsd –r
```

实现系统启动时从板子第一片 FLASH 的最后四个扇区中读取保存的配置信息并在/etc/config 目录下创建生成对应的配置文件。

具体 uClinux/user/flatfsd/中的实现代码如下：

```
if ((rc = flatread(FILEFS)) < 0) {
        if (rc == -5) {
            printf("FLATFSD: non-existent or bad flatfs, "
                "creating new one...\n");
            flatclean();
            if ((rc = flatnew()) < 0) {
                printf("FLATFSD: failed to create new "
                    "flatfs, err=%d errno=%d\n",
                    rc, errno);
                fflush(stdout);
                exit(1);
            }
            sigusr(SIGUSR1);
        } else {
            printf("FLATFSD: failed to read flatfs, err=%d "
                "errno=%d\n", rc, errno);
            fflush(stdout);
            exit(1);
        }
    }
    printf("FLATFSD: created %d configuration files (%d bytes)\n",
        numfiles, numbytes);
    fflush(stdout);
    exit(0);
```

其中关键的实现函数为：flatread。

flatread 函数的功能用英文原文注释为：

Read the contents of a flat file-system and dump them out as regular files.

下面给出其实现代码：

```
int flatread(char *flatfs)
{
    struct flathdr      hdr;
    int         version;
    struct flatent      ent;
    unsigned int        len, n, size, sum;
    int         fdflat, fdfile;
    char        filename[128];
    unsigned char       buf[1024];
    mode_t          mode;
    if (chdir(DSTDIR) < 0)
        return(-1);
    if ((fdflat = open(flatfs, O_RDONLY)) < 0)
        return(-1);
    if (ioctl(fdflat, BMGETSIZEB, &len) < 0)
        return(-2);
    /* 检查头是否合法*/
    if (read(fdflat, (void *) &hdr, sizeof(hdr)) != sizeof(hdr))
        return(-3);
    if (hdr.magic == FLATFS_MAGIC) {
        version = 1;
    } else if (hdr.magic == FLATFS_MAGIC_V2) {
        version = 2;
    } else {
        return(-5);
    }
/* 检查内容是否有效 */
    for (sum = 0, size = sizeof(hdr); (size < len); size += sizeof(buf)) {
```

```
        n = (size > sizeof(buf)) ? sizeof(buf) : size;
        if (read(fdflat, (void *) &buf[0], n) != n)
            return(-4);
        sum += chksum(&buf[0], n);
    }
    if (sum != hdr.chksum)
        return(-5);
    if (lseek(fdflat, sizeof(hdr), SEEK_SET) < sizeof(hdr))
        return(-6);
    for (numfiles = 0, numbytes = 0; ; numfiles++) {
        /* Get the name of next file. */
        if (read(fdflat, (void *) &ent, sizeof(ent)) != sizeof(ent))
            return(-7);
        if (ent.filelen == FLATFS_EOF)
            break;
        n = ((ent.namelen + 3) & ~0x3);
        if (n > sizeof(filename))
            return(-8);
```

读出文件名

```
        if (read(fdflat, (void *) &filename[0], n) != n)
            return(-9);
        if (version >= 2) {
            if (read(fdflat, (void *) &mode, sizeof(mode)) != sizeof(mode))
                return(-7);
        } else {
            mode = 0644;
        }
```

```
    /* Write contents of file out for real. */
```

根据文件名在/etc/config 下创建生成对应文件

```
fdfile = open(filename, (O_WRONLY | O_TRUNC | O_CREAT), mode);
if (fdfile < 0)
        return(-10);
for (size = ent.filelen; (size > 0); size -= n) {
        n = (size > sizeof(buf)) ? sizeof(buf) : size;
```

一次读出一行（n 个字符）

```
        if (read(fdflat, &buf[0], n) != n)
                return(-11);
```

一次写入一行（n 个字符）

```
        if (write(fdfile, (void *) &buf[0], n) != n)
                return(-12);
    }
    /* Read alignment padding */
    n = ((ent.filelen + 3) & ~0x3) - ent.filelen;
    if (read(fdflat, &buf[0], n) != n)
        return(-13);
    close(fdfile);
    numbytes += ent.filelen;
  }
  close(fdflat);
  return(0);
}
```

将配置写入 FLASH 的代码如下：

```
/* Write out the contents of the local directory to flat file-system.
 *   The writing process is not quite as easy as read. Use the usual
 *   write system call so that FLASH programming is done properly.*/
```

```
int flatwrite(char *flatfs)
{
    DIR      *dirp;
    struct stat   st;
    struct dirent *dp;
    struct flathdrhdr;
    struct flatentent;
    unsigned int  sum, size, n, len, pad, total;
    int      fdflat, fdfile, pos;
    unsigned char buf[1024];
    sum = 0;
    pad = 0;
    if (chdir(SRCDIR) < 0)
        return(-1);
```

/* 在 FLASH 上创建文件系统，然后在写入之前先擦除这 4 个扇区的 FLASH */

```
    if ((fdflat = open(flatfs, O_WRONLY | O_CREAT)) < 0)
        return(-2);
    if (ioctl(fdflat, BMGETSIZEB, &len) < 0)
        return(-3);
    if (ioctl(fdflat, BMSGSIZE, &size) < 0)
        return(-3);
    for (pos = len - size; (pos >= 0); pos -= size) {
        if (ioctl(fdflat, BMSERASE, pos) < 0)
            return(-4);
    }
```

在文件头之后写入配置文件的实际内容。这时还没有写入头信息，先要为其预留空间：

```
    if (lseek(fdflat, sizeof(hdr), SEEK_SET) != sizeof(hdr)) //跳到头
```

```
之后
            return(-5);

        /* Scan directory */
        if ((dirp = opendir(".")) == NULL)
            return(-6);

        numfiles = 0;
        numbytes = 0;
        numdropped = 0;
        total = sizeof(hdr);

        while ((dp = readdir(dirp)) != NULL) {
            if ((strcmp(dp->d_name, ".") == 0) ||
                (strcmp(dp->d_name, "..") == 0))
                continue;
            if (stat(dp->d_name, &st) < 0)
                return(-7);
            /*
             *   Write file entry into flat fs. Names and file（文件名＋
内容）
             *   contents are aligned on long word boundaries.（文件内容 4
字节对齐）
             *   They are padded to that length with zeros.
             */
            size = strlen(dp->d_name) + 1;
            if (size > 128) {
                numdropped++;
```

```
            return(-8);
        }
        if ((st.st_size + size + sizeof(ent) + 8) > (len - total)) {
            numdropped++;
            return(-9);
        }
        ent.namelen = size;
        ent.filelen = st.st_size;
        if (write(fdflat, (void *) &ent, sizeof(ent)) != sizeof(ent))
            return(-10);
        sum += chksum((char *) &ent, sizeof(ent));
        total += sizeof(ent);
```

/* 写入文件名，并填充达到 4 字节对齐*/

```
        if (write(fdflat, (void *) dp->d_name, size) != size)
            return(-11);
        total += size;
        sum += chksum(dp->d_name, size);
        size = ((size + 3) & ~0x3) - size;
        if (write(fdflat, (void *) &pad, size) != size)
            return(-12);
        total += size;
```

/* 写入权限 */

```
        size = sizeof(st.st_mode);
        if (write(fdflat, &st.st_mode, size) != size)
            return(-11);
        total += size;
        sum += chksum(&st.st_mode, size);
```

/* 写入配置文件实际内容 */

```
            size = st.st_size;
            if ((fdfile = open(dp->d_name, O_RDONLY)) < 0)
                return(-13);
            for (; (size > 0); size -= n) {
                n = (size > sizeof(buf)) ? sizeof(buf) : size;
                if (read(fdfile, &buf[0], n) != n)
                    return(-14);
                sum += chksum(&buf[0], n);
                total += n;
                if (write(fdflat, (void *) &buf[0], n) != n)
                    return(-15);
            }
            close(fdfile);

            /* Pad to align */
            size = ((st.st_size + 3) & ~0x3)- st.st_size;
            if (write(fdflat, (void *) &pad, size) != size)
                return(-16);
            total += size;

            numfiles++;
            numbytes += ent.filelen;
        }

        closedir(dirp);
        ent.namelen = FLATFS_EOF;
        ent.filelen = FLATFS_EOF;
        if (write(fdflat, (void *) &ent, sizeof(ent)) != sizeof(ent))
```

```
            return(-17);
        sum += chksum((char *) &ent, sizeof(ent));
        /*
         *   Write out the remainder of the FLASH space. If we fill
         *   it with zeros then the chksum won't change...
         */
        memset(&buf[0], 0, sizeof(buf));
        size = len - lseek(fdflat, 0, SEEK_CUR);

        for (;;) {
            n = (size < sizeof(buf)) ? size : sizeof(buf);
            size -= n;
            n = write(fdflat, (void *) &buf[0], n);
            if (n < 0)
                return(-18);
            if (n != sizeof(buf))
                break;
        }
```

```
    /* Construct header, and write that out. */
    //最后才写入头信息
```

```
        hdr.magic = FLATFS_MAGIC_V2;
        hdr.chksum = sum;
        if (lseek(fdflat, 0, SEEK_SET) < 0)  //跳到开始位置
            return(-19);
        if (write(fdflat, (void *) &hdr, sizeof(hdr)) != sizeof(hdr))
            return(-20);
        close(fdflat);
        return(0);
```

```
    }
```

3. 对于比较频繁的数据保存，就要在板子上建立额外的日志型文件系统 JFFS/JFFS2，或者干脆就用 JFFS/JFFS2 取代 ROMFS 作根文件系统。这样板子的目录就是可写的，就像硬盘一样，不需要额外的工具来负责将数据写入 FLASH。JFFS 为 2.0.38 内核所支持，它不支持 JFFS2，JFFS2 到 2.4 内核才被支持，它采用了成熟稳定的 MTD 技术，因此要比 JFFS 稳定。这两种文件系统要在 uClinux 上实现支持并不复杂，但它的实用还需要一些额外的工作，例如烧写工具的配合，新型 image.bin 编译生成，因为真正产品化的软件是不能允许每次启动后都还要进行许多的手工操作，例如加载文件系统等，板子出厂烧写也要一次完成，而不能还要分多次烧写等等，这些工作都是比较繁杂的，而且没有烧写工具的源代码是无法完成的。

下面介绍 JFFS2 的实现：

JFFS2 首先要内核支持：在 uClinux 下执行 make menuconifg 中选择 MTD，并选择其中的正确选项。还有在 file systems 中选择 JFFS2。下面列表给出相关的内核选项：

```
[*] Memory Technology Device (MTD) support
    [ ] Debugging??
    [*]     MTD partitioning support
    [ ]     RedBoot partition table parsing??
    --- User Modules And Translation Layers??
    [*]     Direct char device access to MTD devices
    [*]     Caching block device access to MTD devices
    [ ]     FTL (Flash Translation Layer) support
    [ ]     NFTL (NAND Flash Translation Layer) support
  在：RAM/ROM/Flash chip drivers    --->中：
  [*]     Detect flash chips by Common Flash Interface (CFI) probe
    [ ]     Detect non-CFI AMD/JEDEC-compatible flash chips
    [ ]     Flash chip driver advanced configuration options
```

[ ] Support for Intel/Sharp flash chips
[*] Support for AMD/Fujitsu flash chips
[*] Support for RAM chips in bus mapping
[*] Support for ROM chips in bus mapping
[ ] Support for absent chips in bus mapping
[ ] Older (theoretically obsoleted now) drivers for non-CFI chips

在 Mapping drivers for chip access --->中：

[ ] CFI Flash device in physical memory map
[*] CFI Flash device mapped on Lineo SecureEdge (uClinux)
[ ] CFI Flash device mapped on Key Technology devices
[ ] Hitachi SH KeyWest Intel flash device

其它都不再选择。

在 File systems --->中：

[*] Journalling Flash File System v2 (JFFS2) support
(0) JFFS2 debugging verbosity (0 = quiet, 2 = noisy)

在 Block devices --->中：

[ ] Normal PC floppy disk support
[ ] XT hard disk support
[ ] Parallel port IDE device support
[ ] Loopback device support
[ ] Network block device support
[*] RAM disk support
(4096) Default RAM disk size
[ ] Initial RAM disk (initrd) support
[ ] ROM disk memory block device (blkmem)

下一步，修改 linux-2.4.x/drivers/mtd/maps/nettel-uc.c，使 ROOT_DEV 对应 mtd 分区之一。当然首先要在 nettel-uc.c 中规划自定义的 FLASH 分区。

```
static struct mtd_partition nettel_4mb[] = {
        { name: "Bootloader", offset: 0x00000000, size:   0x00004000 },
        { name: "Bootargs",   offset: 0x00004000, size:   0x00002000 },
        { name: "MAC",        offset: 0x00006000, size:   0x00002000 },
        { name: "Spare",      offset: 0x00008000, size:   0x00008000 },
        { name: "Kernel",     offset: 0x00010000, size:   0x000A0000 },
        { name: "Image",      offset: 0x000B0000, size:   0x00150000 },
        { name: "Flash",      offset: 0x00000000, size:   0x00200000 },
        { name: "Image2",     offset: 0x00200000, size:   0x00200000 },
        { name: "Flash2",     offset: 0 }
};
```

在这里我们必须解释一下所谓文件系统的一些概念：

首先 LINUX 下所有的文件系统都是要加载（mount）使用的，所不同的是根文件系统是在内核启动时 mount 的，其它的文件系统都是在应用态使用应用程序 mount 加载的，一般的就放在启动脚本/etc/rc 文件中进行加载。例如：

```
/bin/expand /etc/ramfs.img /dev/ram0
/bin/expand /etc/ramfs.img /dev/ram1
mount -t proc proc /proc
mount -t ext2 /dev/ram0 /var
mount -t ext2 /dev/ram1 /etc/config
mount -t jffs2 /dev/mtdblock7 /sbin
```

其次，文件系统都是 MEMORY（FLASH/SDRAM）中的存储块，这些存储块都是用对应工具生成的 IMAGE 映像文件烧写到 FLASH 对应的位置或者展开到 RAM 中形成的，例如 RAM 盘用 mke2fs 工具，ROMFS 用 genromfs 工具，JFFS 用 mkfs.jffs 工具，JFFS2 用 mkfs.jffs2 工具。一般的，

都是针对一个目录来使用这些工具，将整个目录转换成一个 IMAGE 映像文件，则生成的 IMAGE 文件内部就包含了该目录的存储结构信息，一旦通过 mount 被加载，IMAGE 里面的文件目录结构信息就恢复出来，从而在板子上引入新的文件系统。

一般的，文件系统的IMAGE文件都是烧写在FLASH上指定的位置，所谓加载就是通知内核这个IMAGE映像文件在FLASH上的起始地址，从而保证可以正确的恢复出里面保存的文件目录结构。ROMFS是通过blkmem.c中的arena[0].address来指示的，JFFS/JFFS2则是通过FLASH分区表来指示的。例如上面我们给出的例子里有：

```
{ name: "Image2",     offset: 0x00200000, size:   0x00200000 }
```

它是MTD_PARTITION NETTEL_4MB分区表中的第7个分区，对应的设备就是MTDBLOCK7，因此/ETC/RC中：

```
mount -t jffs2 /dev/mtdblock7 /sbin
```

就是要寻找 mtdblock7 对应的第七个 FLASH 分区的起始地址，即：

FLASH 的起始地址＋第七个分区的 offset＝

0xffc00000+0x00200000=0xffe00000

其实就是把整个第二片 FLASH 的 2M 空间全部作为第七个分区了，也就是板子启动后的/sbin/目录就是板子的整个第二片 FLASH 空间，它可读可写，是个 JFFS2 的文件系统。

同时，这种实现机制的另外一个要求就是在烧写板子 FLASH 的时候，烧在板子第二片 FLASH 上的必须是一个 JFFS2 文件系统的映像文件。这个文件用 mkfs.jffs2 工具生成：

mkfs.jffs2 -q -b -r jffs2_sbin --pad= 0x200000 -o jffs2_sbin.bin

因为指定了--PAD= 0X200000，所以生成的 JFFS2_SBIN.BIN 大小就是固定的 2M 字节，即第二片 FLASH 的整片字节映像。其中 JFFS2_SBIN 目录下放置了用户想在板子启动后/SBIN/目录下放置的应用程序。

下面就需要将 JFFS2 文件系统的映像文件 JFFS2_SBIN.BIN 和原来编译生成的 IMAGE.BIN 合并起来生成新的 IMAGE.BIN 以供烧写。这里我们定制了自己的映像文件合并生成工具 GEN_IMAGE，其使用方法如下：

gen_image image.bin jffs2_sbin.bin

其源代码如下：

```
    int main(int argc,char *argv[])
    {
        int oimage;
        int linux;
        int jffs2_sbin;
        unsigned char buffer[1024];
        int filesize = 0;
        int n=4096;//Fill the two FLASH,total 4MB
        struct stat *filestat;
        ////////////////////////////////////////////////////////////////////
////
        //Fill the 4M image.bin with 0xFF
        printf("Fill the 4M image.bin with 0xFF!\n");
        remove("/tftpboot/image.bin");

        memset(buffer,0xff,sizeof(buffer));
        if((oimage = open("/tftpboot/image.bin",O_CREAT|O_RDWR)) < 0){
            printf("Open output failed.\n");
            exit(0);
        }
        else{
            while(n--)
```

```
                write(oimage,buffer,sizeof(buffer));
            lseek(oimage,0,SEEK_SET);
        }
        printf("Padding finished!\n");
        ///////////////////////////////////////////////////////////////////
////
        //open original image.bin first
        if((linux = open(argv[1],O_RDONLY)) < 0){
            fprintf(stderr,"Open file image.bin error %d\n",errno);
            exit(1);
        }
        //jffs2_sbin.bin
        if((jffs2_sbin = open(argv[2],O_RDONLY)) < 0){
            fprintf(stderr,"Open file jffs2_sbin.bin error %d\n",errno);
            exit(1);
        }else{
            filestat = (struct stat*)malloc(sizeof(struct stat));
            fstat(jffs2_sbin,filestat);
            printf("file %s size %d\n",argv[2],filestat->st_size);
            free(filestat);
        }

        //write original image.bin
        while(n = read(boot_code,buffer,sizeof(buffer)))
            write(oimage,buffer,n);

        //write jffs2_sbin.bin to 0x200000
        lseek(oimage,0x200000,SEEK_SET);
```

```
        while(n = read(jffs2_sbin,buffer,sizeof(buffer)))
            write(oimage,buffer,n);

        fchmod(oimage,S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH);
        close(oimage);
        close(linux);
        close(jffs2_sbin);
        exit(0);
    }
```

将这个工具整合到 uClinux 的 Makefile 中去，例如加到 uClinux/vendors/HHTECH/M5272/Makefile 中，则最后生成的 /tftpboot/image.bin 文件为整 4M 字节大小，用烧写工具烧制到板子 FLASH 上，系统启动后则板子上/sbin/目录为 JFFS2 文件系统，可读可写，再除去 /var/和/tmp/目录为 RAM 盘，/proc 为 PROC 文件系统外，其它文件和目录都是 ROMFS 文件系统，只读不可写，至此就实现了 JFFS2 文件系统。

下面给出一些注意事项：

1、在 romfs/dev/下用 touch @mtdblock0,b,31,0 创建 JFFS2 文件系统所需的设备文件时，还必须修改 vendors/HHTECH/M5272/Makefile，删除其中创建 rom0～9，因为它的 MAJOR 和 mtdblock 冲突，都是 31。

2、在宿主机上 uClinux/romfs/目录下必须手工创建 sbin 目录：

mkdir sbin

这个目录本身是空的，但系统启动后，将 JFFS2 文件系统加载到/sbin/目录上之后，它里面就有了 JFFS2 文件系统的内容。

3、若要整个废弃 ROMFS，整个根文件系统也采用 JFFS2 的话，必须修改 linux-2.4.x/init/main.c 中：

mount_flag=0;

原来默认的是 MS_RDONLY，这样整个文件系统才可读可写。

4、 JFFS2 至少需要 5 个 FLASH 扇区来作“垃圾收集”工作，所以

要有个可写的 JFFS2 目录，就必须给它分配多于 5 个的扇区数，例如前面 mtd_partition nettel_4mb 中有效的分区，即第七个分区”**Image2**”，它的 size 必须大于 5 个扇区，即大于 0x50000 才能真正的实现一个 JFFS2 文件系统目录。

2.5 调试内核驱动或者应用程序技巧

在程序调试阶段会频繁的修改程序，然后重新下载进去，由于烧写速度狠慢而下载速度相对较快，可以对内核进行一下修改使得下载到 RAM 中的应用程序可以直接跑而不需烧写到 FLASH 中去：

把 linux-2.4.x/drivers/block/blkmem.c 中

```
#define FIXUP_ARENAS        arena[0].address = 0x000d0000;
```

一句改为

```
#define FIXUP_ARENAS        arena[0].address = 0x0c400000;
```

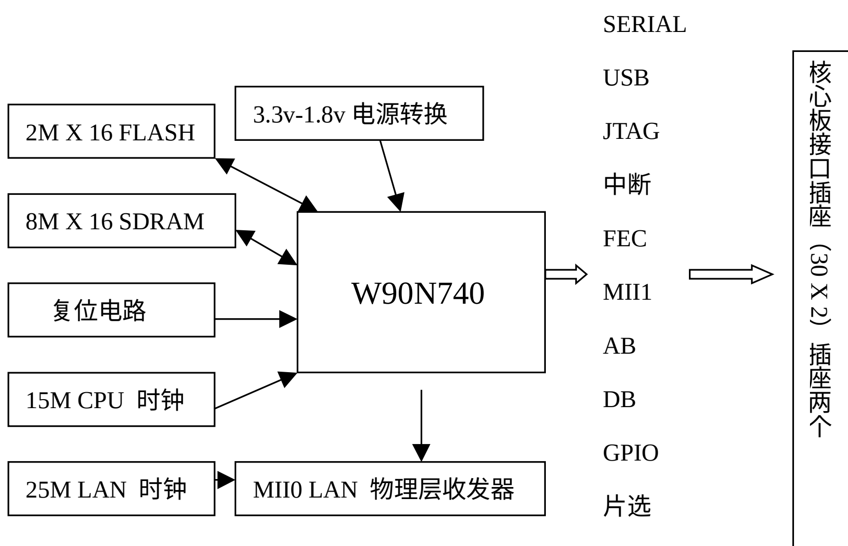
则内核启动时从 0x0c400000，即 romfs 下载到内存中的位置 mount 文件系统。

而对于内核来说，下载完可以直接 boot，无需烧写固化。

第三章 硬件系统

3.1 功能模块结构图

核心板:



3.2 各个部分的构成

- 内存部分：1 片 16×1M 位数据宽度的 FLASH，共 2M 字节 Flash（AM29LV320，如有不同型号，则是完全兼容的器件），速度 90ns；两片 4M×16 位数据宽度的 SDRAM（HY 57V641620，如有不同型号，则是完全兼容的器件）构成，共 16M 字节 SDRAM。
- 数据通讯口和外部输入输出口：底板提供 2 个 RS232 接口；

3.3 内存映射

华恒开发板两片 4M ×16 位数据宽度的 SDRAM 构成，共 16M 字节 SDRAM，从 0x0c00_0000----0x0cff,ffff

1 片 2M FLASH，从 0x7F00_0000----0x7F3f_ffff

Am29LV160DB FLASH 扇区（sector）分布：

2M 字节一片 FLASH，工作在双字节模式，共 35 个扇区，除前 8 个扇区不规则，大小 8K 外，剩余扇区均为 64K 字节大小。具体请参见 AMD 数据手册 AM29LV160D.pdf

0x0, 0x4000, 0x2000, 0x2000, 0x8000, 0x10000,
0x20000, 0x30000,0x40000,0x50000,0x60000,
0x70000, 0x80000, 0x90000, 0xa0000,0xb0000,
0xc0000, 0xd0000, 0xe0000, 0xf0000,0x100000,
0x110000, 0x120000, 0x130000, 0x140000, 0x150000,
0x160000, 0x170000, 0x180000, 0x190000, 0x1a0000,
0x1b0000, 0x1c0000, 0x1d0000, 0x1e0000, 0x1f0000,

3.3 总线

- W90N740 是 24 位地址总线和 32 位数据总线。80M 的主频和 66M 的总线速度。W90N740 所用的 ARM720 的核内部使用的是 32 位的内部地址总线，但其中只有 24 位通过 CPU 引脚给出来。
- 若外接 8 位或 16 位数据宽度的外设芯片，与 CPU 相接时，740 的数据总线宽度是可配置的，可分别配为 32 位、16 位或 8 位模式。

3.4 片选分配

片选	接口	备注
GCS0	FLASH	16 位模式
NSCS0	SDRAM1/SDRAM2	2 片 16 位拼做 32 位使用，共用一个片选

3.5 中断分配

IRQ0	MII
IRQ1	未用
IRQ2	未用
IRQ3	未用

cat /proc/interrupts

```
  2:          0    <NULL>
  6:         98    serial
  7:      12814    timer
  9:          0    usb-ohci
 13:          0
 14:          0
 15:          0
 16:        121
Err:          0
```

3.6 GPIO 使用情况

GP20-17	中断
GP16，15	DMA
GP14-12	时钟
GP11-GP4	串口
GP3	RY0
GP2-GP0	未用

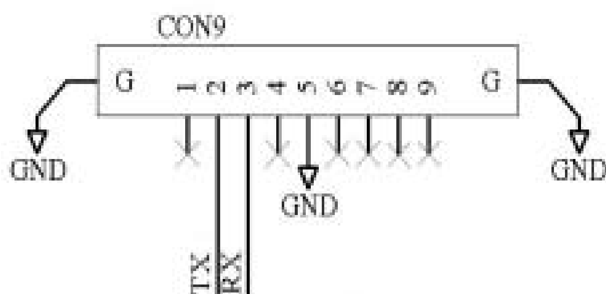
3.7 接口管脚定义

核心板图如下所示：

1. LED 指示灯

状态指示部分：由 LED 提供一个电源指示；一个 FLASH 状态指示；

2. RS232 UART0 接口：【DB9 孔头】



管脚说明：

TX0	串行口 0 发送线
RX0	串行口 0 接受线
RTS0	串行口 0 请求发送线
CTS0	串行口 0 清除发送线

3. 两个 60 线插座（CN1，CN2）将 W90N740B 处理器上几乎所有的信号引出。

其中 CN1 插座（CON120）如下页所示：

管脚说明：

A[0..23]:地址线

D[0..31]：数据线

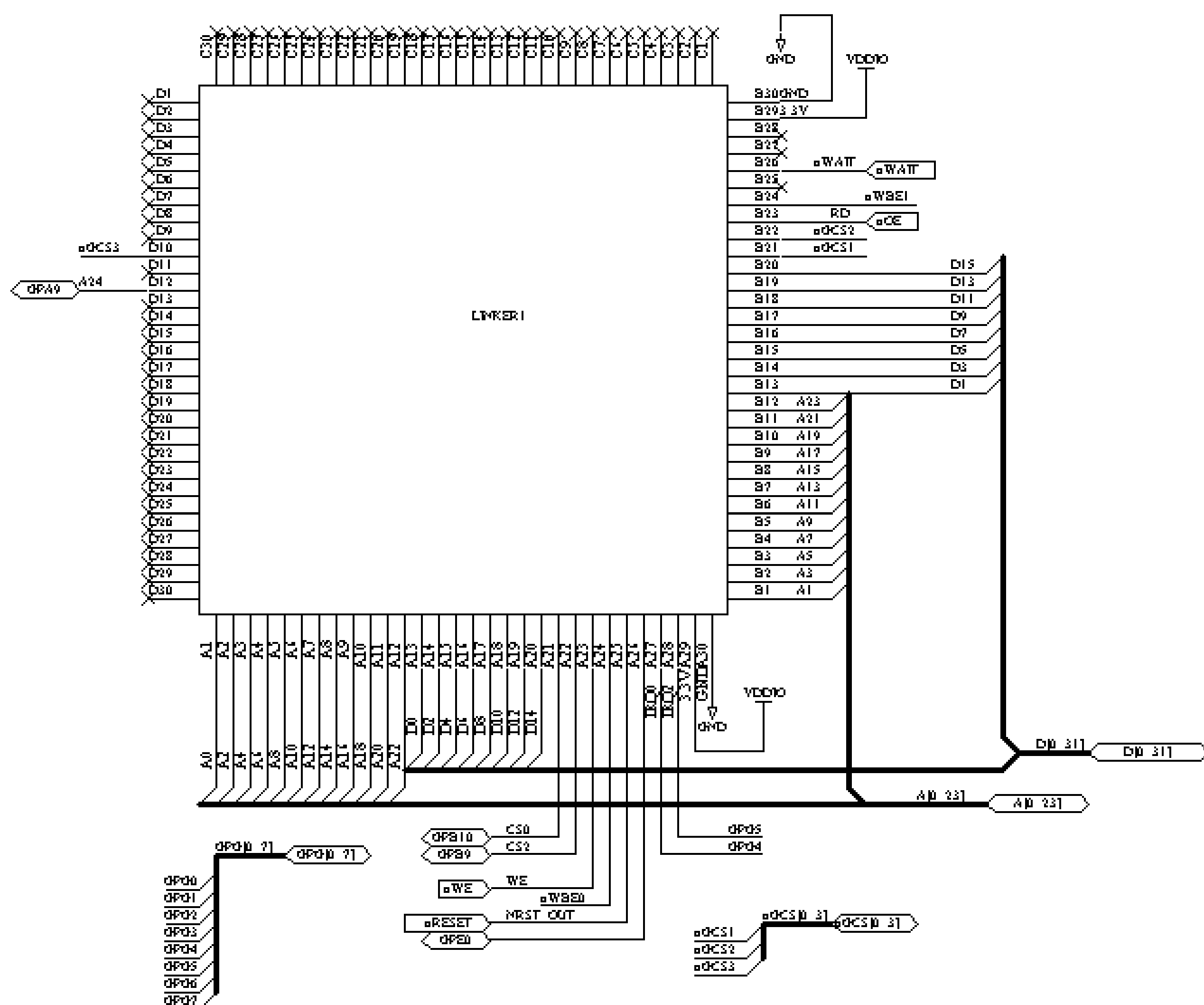
GPX:通用 I/O 口

nGCS[0..3]：通用片选线

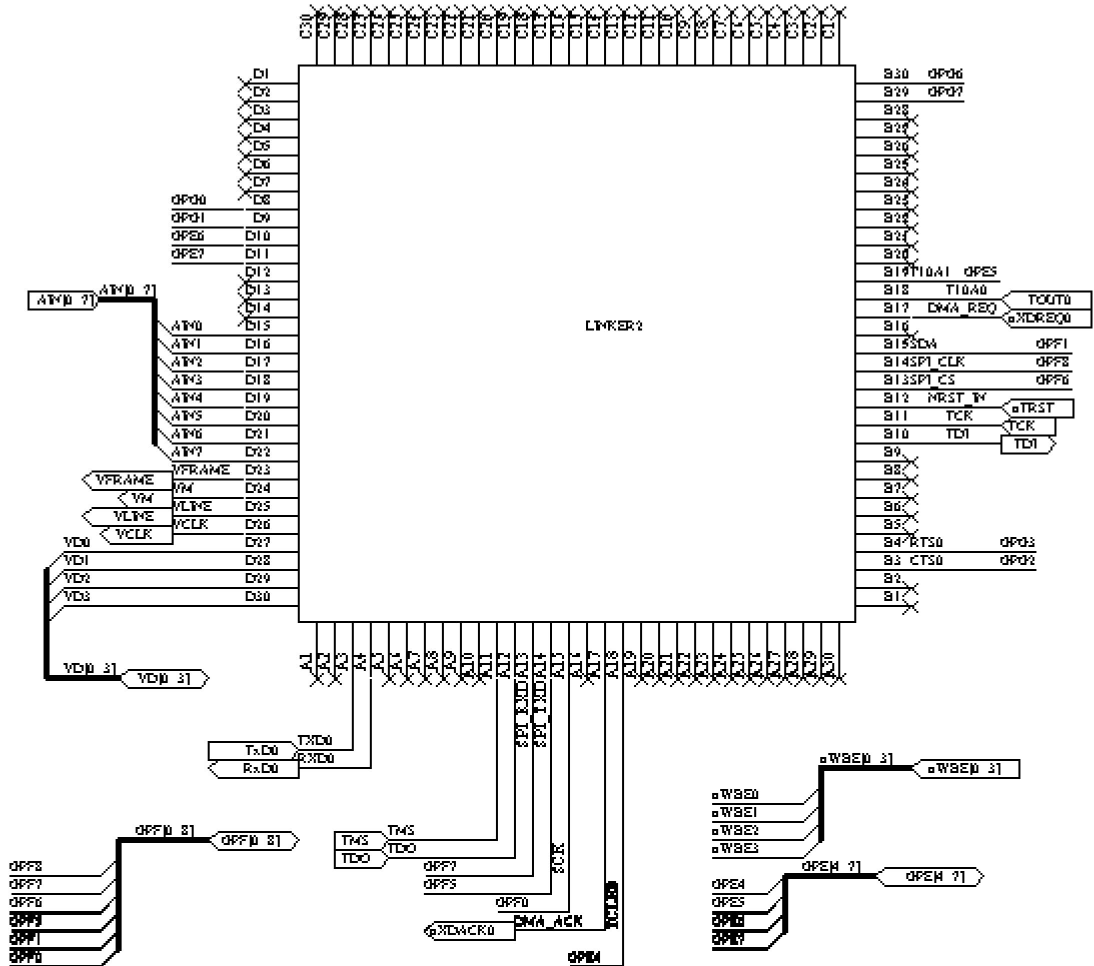
nWE/nRD:写读线

nRESET/nWAIT：复位与等待

nWBE0/1：字节操作选择线



CN2 插座（CON120）如下所示：



管脚说明：

GPxx:通用 I/O 口

VD[0..3]：LCD 数据信号(注意 W90N740 的以太网 MAC 器数据线为[0..7], 因为管脚复用，其它信号线用其他名称表示；别的端口有类似情况)

VLINE/VM/VFRAME/VCLK：以太网 MAC 信号与时钟信号。

AIN[0..7]：模拟信号输入端

RXD/TXD/RTS/CTS：UART 接口信号

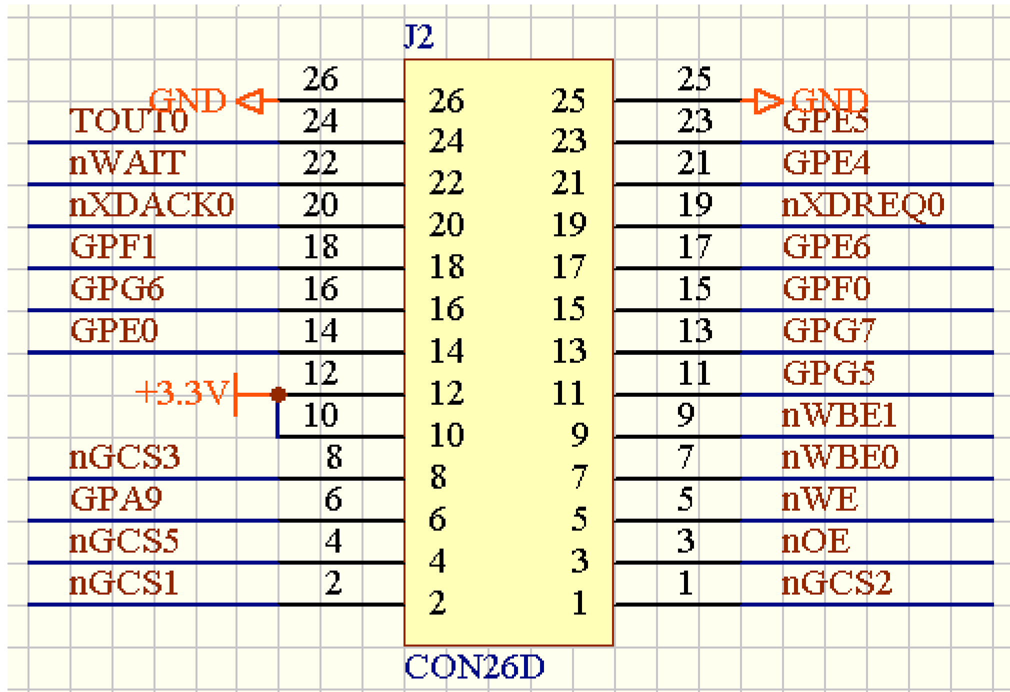
TMS/TDO/TDI/TCK/Ntrst：JTAG 接口信号

底板上提供了 J1 和 J2 两个插座，其中 J1 提供了所有的 AB/DB，J2 则引出了所有系统剩余的信号引脚资源。

两个插座上的信号定义参见下图：

J1
40PIN

Signal	Pin
A0	1
A1	2
A2	3
A3	4
A4	5
A5	6
A6	7
A7	8
A8	9
A9	10
A10	11
A11	12
A12	13
A13	14
A14	15
A15	16
A16	17
A17	18
A18	19
A19	20
A20	21
A21	22
A22	23
A23	24
D0	25
D1	26
D2	27
D3	28
D4	29
D5	30
D6	31
D7	32
D8	33
D9	34
D10	35
D11	36
D12	37
D13	38
D14	39
D15	40



特别说明：由于 W90N740 的大部分管脚具有多种复用功能，而 HHARM740 几乎将处理器所有的资源都引到了底板上。所以对于开发者来说，对系统进行不同的寄存器配置，可将这些管脚作为不同的功能。

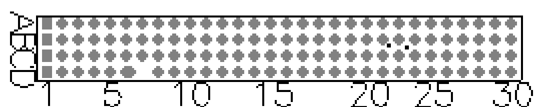
电气特性：

电源	+5V——+12V
工作环境温度(注 1)	0°C ~ 40°C
保藏温度	－20°C ~ 70°C
相对湿度	25%~95%

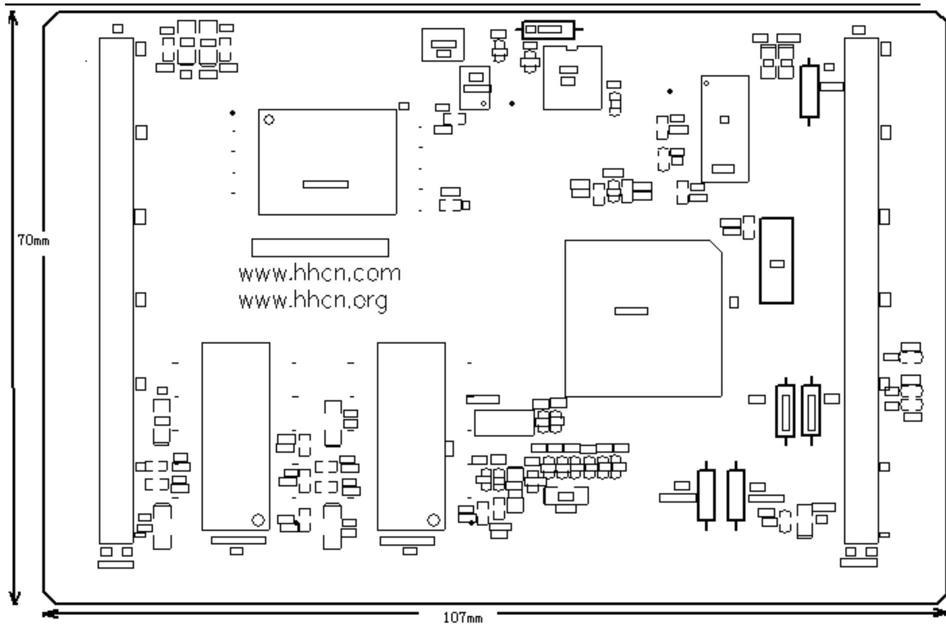
第四章 机械特性

核心板 PCB 上有两个 4 排孔的插座，实际上只用到了 A、B 两排，所以在核心板上可能焊接的不是 4 排孔插座：

CN1(2)插座图示如下：



核心板 HHARM740 正面图示如下：



底板如下图示：

第五章 底板的硬件设计

5.1 基本板的设计

基本板包括如下主要部分:

- 与核心板的连接器（由 2 个 2 X 30 的双列插座构成）；
- 基本端口，包括两个串口、JTAG 口；
- 电源：将外部输入的未稳压电源稳定为核心板和基本板上芯片所需的 3.3V 电源。

由上面的“核心板连接器一、二”的引出线排布可以看到，在核心板的 PCB 设计中，由于我们使用的是脚间距为 2.0mm 的双排插座（针），从系统走线和核心板电磁兼容性上考虑，我们将“核心板连接器一、二”的外侧引脚大部分作为电源线和地线的引出脚。

对于串行口 1/2，因为在不同的应用中可能会被作为 RS232/RS485/422，因此对该口必须使用相应的电平转换芯片。在基本板的设计中我们给出的是作为 RS232 端口的方案。

5.2 用户底板原理性设计和硬件方案制定

5.2.1 基本端口的设计

以太网接口、串口、JTAG 口的设计可以直接参考基本板的设计，具体电路请参考基本板的原理图和 PCB。另外必须注意的是，在串行口电平转换芯片的选择和使用中必须注意电平兼容问题，在基本板中我们使用的是 3.3V 工作电压的 RS232 电平转换芯片。

5.2.2 电源的设计

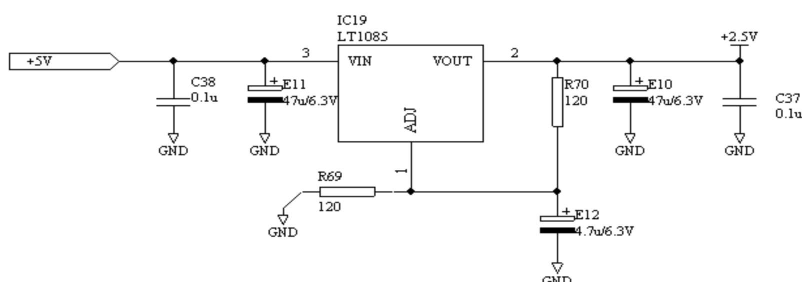
HHARM740 接口底板的工作电源为+3.3V，由于电源消耗功率较小，因此我们使用的是较为廉价的 LT1085 线性稳压芯片，使用基本板的上下面铜箔作为散热面，并且使用 5V--12V 直流电源供电。

核心板工作电源为双直流电源 3.3V 和 2.5V，其中 3.3V 由底板直接供给，2.5V 由 3.3V 通过 LT1764EQ-2.5 变换得到。

在其他的应用设计中根据不同的电源消耗需求，可以选择线性稳压源方案和开关稳压源方案。对于前一种选择，可以获得低噪声、廉价等益处，但同时也有效率低，发热较大等缺点；对于开关电源方案正好与线性电源的优缺点相反。

一般的，使用线性电源时，我们给出以下两种参考：

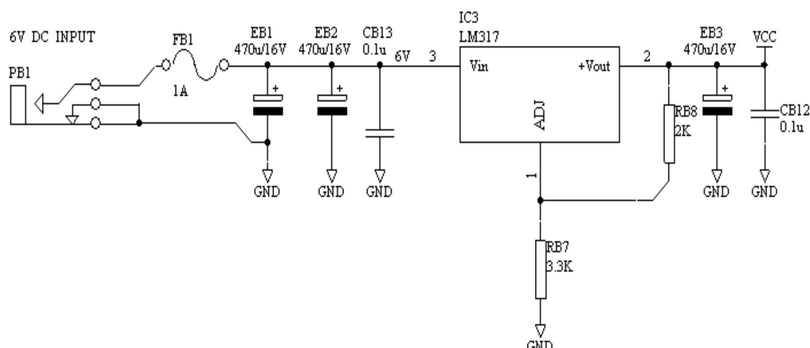
其一参见下图：



这是一个使用低压差线性稳压器获得 2.5V 稳压输出的例子，输出电压的计算可以使用以下的公式：Vo=1.25V * (R69+R70)/R70；芯片上的耗散功率可以使用以下公式计算：Pc=(Vi-Vo)*Io，由上可以看到，除了输出电流，输入和输出间的压差正比于芯片上的耗散功率。

稳压芯片的金属背板表面工作温度必须小于 80 摄氏度，设其金属背板到散热器的热阻为 Rt1，散热器到空气的热阻为 Rt2，则稳压芯片金属背板与环境的温升为：ΔT=Pc/(Rt1+Rt2)；由此可以看到减少温升必须减小 Pc 或者总热阻，后者可以通过减小 Rt1（使用导热硅脂于散热器于管壳之间），减少 Rt2（增大散热器面积）；

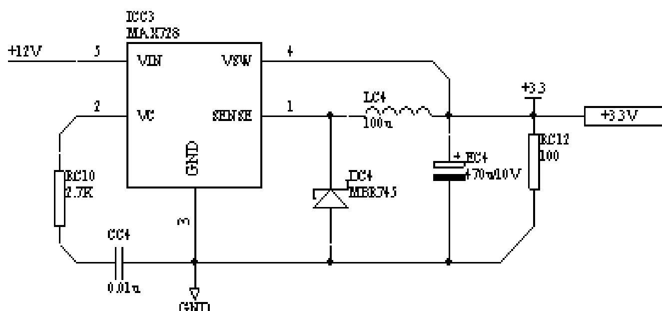
其二参见下图：



这是一个使用最廉价的普通线性稳压器获得 3.3V 稳压输出的例子，输出电压的计算可以使用以下的公式：Vo=1.25V * (RB8+RB7)/RB8；其他同上，但要注意的是 LM317 必须有大于 2.5V 的输入输出压差才能工作在稳压状态。因此最下输入电压要大于上者。

使用开关电源时，我们给出以下参考：

见下图：



要注意的是电感 LC4 的选择，其最小平均直流工作电流必须大于输出电流(既在规定的输出电流下其不能出现磁芯饱和现象)。另外**必须注意 PCB 的排版**，这将决定电源的噪声幅度，在以上电路中 EC4 为钽电解电容，在一般的应用中必须另外附加并联一个 0.1u 的瓷介电容,以率除高频分量。

电源类型的选择，最终是效率、成本、体积等因素的综合考虑。

5.2.3 其它电路部分的设计

在这里特别指出应当充分重视对核心板各个引出线添加缓冲电路，以充分保证核心板和处理器的稳定性；对于核心板连接器的地线和电源线引脚应当全部连接到相应的地和电源上。

5.2.4 PCB 设计和排版时的注意事项

PCB 布局时的考虑

首先由于 4 X 30PIN 的 2.0mm 间距双列插座的脚比较密，在焊盘间穿线不是很方便，在核心板设计中将上述插座的外围引脚大部分作为电源或地线。其次由于核心板的读写访问速率较高，因此必须考虑到布线传输延迟对电路的影响。在底板 PCB 走线设计中时应当将涉及到数据总线、地址总线、内存访问控制信号和中断信号等的电路部分排布在 CN1,CN2 连接器之间，以方便布线和减少信号延迟。

简单应用时可以使用 2 层板，并且大面积铺地；复杂的应用使用多层布线，并且至少有一层为地层（最好将相邻的层作为电源层。当板上有数个电源品种时，一般来讲如果可以将使用同种电源的电路部分布局在临近区域内，然后将电源层分割为数个区块。）

另外还应当注意以下事项：

建议保留 JTAG 端口；

核心板接口插座的整体布局；

板上提供必要的测试焊盘；

5.2.5 PCB 在电路稳定性和抗干扰方面的考虑

应重点考虑以下几点：

1．　合理的地层和电源层分割

建议使用较为完整的一个 PLANE 作为地层，如果存在多个“地”(例如：数字地，模拟地，保护地等）可以在同一个地层上适当分割获得。对于电源层类似上述，并且应当注意低噪声电路部分电源和高噪声电路部分电源的隔离或滤波。

2．　合理的去耦电容排布

在电源进线端，在各个数字芯片电源引脚附近等分布适当的去耦电容。对核心板的隔离，以及在各个端口使用适当的输入输出缓冲；

第六章 售后服务及技术支持

在华恒嵌入式 Linux 开发套件上进行开发时所遇到的一切问题，均可在华恒嵌入式 linux 技术论坛： http://www.hhcn.org 站点上得到专家们快捷、准确的回答。

另外，还可以发 mail 到华恒公司工程师组电子信箱 support@hhcn.com，或者直接致电华恒公司，您都会得到及时的答复。

http://www.hhcn.org 上还提供支持华恒嵌入式 linux 开发板应用开发的技术资料、源代码、开发源代码的 GNU 项目以及开发套件软件的升级版本和补丁，使得基于华恒嵌入式 linux 开发板进行产品开发更快捷、便利。

华恒嵌入式 Linux 开发套件 ,整体保修期为六个月，第一个月可免费更换，但电源、插座、JTAG 卡及电缆等易耗件及人为损坏不在保修范围之内。

警告：

请务必注意静电的防护。超过任何最大承受值，均会对产品产生永久损害。同时，不推荐在临界状态使用产品。

附录 A LINUX 常见术语

LINUX

Linux 是一种 UNIX 操作系统的克隆，它（的内核）由 Linus Torvalds 以及网络上组织松散的黑客队伍一起从零开始编写而成。 Linux 的目标是保持和 POSIX 的兼容。

Linux 具备现代一切功能完整的 UNIX 系统所具备的全部特征，其中包括真正的多任务、虚拟内存、共享库、需求装载、共享的写时复制程序执行、优秀的内存管理以及 TCP/IP 网络支持等。

Linux 的发行遵守 GNU 的通用公共许可证。

Linux 起初为基于 386/486 的 PC 机开发，但现在，Linux 也可以运行在 DEC Alpha、SUN Sparc、M68000，以及 MIPS 和 PowerPC 等计算机上。

POSIX

POSIX 表示可移植操作系统接口（Portable Operating System Interface，缩写为 POSIX 是为了读音更像 UNIX)。电气和电子工程师协会（Institute of Electrical and Electronics Engineers，IEEE）最初开发 POSIX 标准，是为了提高 UNIX 环境下应用程序的可移植性。然而，POSIX 并不局限于 UNIX。许多其它的操作系统,例如 DEC OpenVMS 和 Microsoft Windows NT，都支持 POSIX 标准，尤其是 IEEE Std. 1003.1-1990（1995 年修订）或 POSIX.1，POSIX.1 提供了源代码级别的 C 语言应用编程接口（API）给操作系统的服务程序，例如读写文件。POSIX.1 已经被国际标准化组织（International Standards Organization，ISO）所接受，被命名为 ISO/IEC 9945-1:1990 标准。

POSIX 现在已经发展成为一个非常庞大的标准族，某些部分正处在开发过程中。POSIX 与 IEEE 1003 和 2003 家族的标准是可互换的。

GNU

GNU 是 GNU Is Not UNIX 的递归缩写，是自由软件基金会的一个项目，该项目的目标是开发一个自由的 UNIX 版本，这一 UNIX 版本称为 HURD。尽管 HURD 尚未完成，但 GNU 项目已经开发了许多高质量的编程工具，包括 emacs 编辑器、著名的 GNU C 和 C++ 编译器（gcc 和 g++），这些编译器可以在任何计算机系统上运行。所有的 GNU 软件和派生工作均适用 GNU 通用公共许可证，即 GPL。GPL 允许软件作者拥有软件版权，但授予其他任何人以合法复制、发行和修改软件的权利。
Linux 的开发使用了许多 GNU 工具。Linux 系统上用于实现 POSIX.2 标准的工具几乎都是 GNU 项目开发的，Linux 内核、GNU 工具以及其他一些自由软件组成了人们常说的 Linux：
符合 POSIX 标准的操作系统 Shell 和外围工具。
C 语言编译器和其他开发工具及函数库。
X Window 窗口系统。
各种应用软件，包括字处理软件、图象处理软件等。
其他各种 Internet 软件，包括 FTP 服务器、WWW 服务器等。
关系数据库管理系统等。

GPL

GPL (General Public License)
GPL 的文本保存在 Linux 系统的不同目录下的命名为 COPYING 的文件里。例如，键入 cd /usr/doc/ghostscript* 然后再键入 more COPYING 可查看 GPL 的内容。
GPL 和软件是否免费无关，它主要目标是保证软件对所有的用户来说是自由的。GPL 通过如下途径实现这一目标：

它要求软件以源代码的形式发布，并规定任何用户能够以源代码的形式将软件复制或发布给别的用户。
它提醒每个用户，对于该软件不提供任何形式的担保。

如果用户的软件使用了受 GPL 保护的任何软件的一部分，那么该软件就继承了 GPL 软件，并因此而成为 GPL 软件，也就是说必须随应用程序一起发布源代码。

GPL 并不排斥对自由软件进行商业性质的包装和发行，也不限制在自由软件的基础上打包发行其他非自由软件。

遵照 GPL 的软件并不是可以任意传播的，这些软件通常都有正式的版权，GPL 在发布软件或者复制软件时声明限制条件。但是，从用户的角度考虑，这些根本不能算是限制条件，相反用户只会从中受益，因为用户可以确保获得源代码。

尽管 Linux 内核也属于 GPL 范畴，但 GPL 并不适用于通过系统调用而使用内核服务的应用程序，通常把这种应用程序看作是内核的正常使用。

Linux 的主要发行版

Red Hat Linux

采用 RPM 的软件保管理方式，软件的安装、卸载和升级非常方便，并提供了大量的 图形化管理工具，是初学者的最佳选择。

Mandrake

Slackware

Debian GNU/Linux

是由 GNU 发行的 Linux 版本，最符合 GNU 精神。提供了最大的灵活性，适合 Linux 的高级用户。

附录 B 常用 LINUX 命令

以下均以 REDHAT LINUX 为例说明。

1) 基本命令：

ls：显示当前目录下的所有文件和目录。

ls –a ：可以看到隐藏的文件，如以.开头的文件。

pwd：显示当前目录路径。

ps：列举当前 TTY 下所有进程

ps –A：列举所有

cd 目录名：进入目录

mkdir 目录名：创建目录

rmdir 目录名：删除空目录

rm –rf 目录名：强行删除整个目录内容（无法恢复），其中 f 表示强制不进行提示，r 表示目录递归。

2) LINUX 下的文件和目录是区分大小写的。

3) TAB 文件目录匹配搜索的使用

例如华恒软件安装的目录为：/HHARM740，假设/目录下没有其它以 HH 字符开头的其它目录和文件，则要进入这个目录，只需敲入：

cd /HH

然后按下 TAB 键，则 SHELL 会自动匹配找到 HHARM740 目录，这样就不必完全键入剩余的 ARM740-R1 字符，这个功能在访问名字很长的文件和目录时非常有效，可以大大提供键盘输入的速度，极为方便。

4) ncftp 工具的使用：

ncftp 是 LINUX 下非常好的 FTP 工具软件，它除了支持 FTP 命令操作外，还支持 LINUX SHELL 下的命令用法，例如，它也支持 TAB 键用法，支持目录上传和下载（用-r 或-R 参数）。ncftp 的用法，例如要 FTP

一台 IP 为 192.168.2.32 的 LINUX PC 机 A，命令如下：

ncftp –u hhcn 192.168.2.32

其中 hhcn 为 A 机器上的合法的用户，连接上之后会提示输入 hhcn 用户的密码，密码验证通过后，就进入 ncftp 命令提示符。

5) 编程时获取帮助 man（类似于 VC 编程中的 MSDN）

man，即 manunal，是 UNIX 系统手册的电子版本。根据习惯，UNIX 系统手册通常分为不同的部分（或小节，即 section），每个小节阐述不同的系统内容。目前的小节划分如下：

1. 命令：普通用户命令
2. 系统调用：内核接口
3. 函数库调用：普通函数库中的函数
4. 特殊文件：/dev 目录中的特殊文件
5. 文件格式和约定：/etc/passwd 等文件的格式
6. 游戏。
7. 杂项和约定：标准文件系统布局、手册页结构等杂项内容
8. 系统管理命令。
9. 内核例程：非标准的手册小节。

手册页一般保存在 /usr/man 目录下，其中每个子目录（如 man1, man2, ..., manl, mann）包含不同的手册小节。使用 man 命令查看手册页。

常用 man 命令行：

man strtoul

6) 取消 root 密码：

vim /etc/shadow

可以看到第一行内容大致如下：

root:1dVVd5YVP$OgZG58TL/NRExTfcr6URH.:11829:0:99999:7:-1:-1:134539236

要取消 root 密码，只需将第一行 root 后第一对:之间的字符全部删除即可，删除后如下：

root::11829:0:99999:7:-1:-1:134539236

然后用:w!强行存盘（因为 shadow 文件是只读的）后用:q 退出 vi 则实现取消了 root 密码。

7) 修改 PC 机 IP 地址：

ifconfig eth0 192.168.2.32

8) 压缩/解压缩：

LINUX 的软件一般是以.gz 或.tar 或者.tar.gz 结尾的。前者是由 gzip 压缩的，后者是先用 tar 归档，在用 gzip 压缩而成的。

1、以.gz 结尾的为压缩文件，用命令：gzip -d filename 来解压，得到的文件在当前目录中，但已没有了.gz。

2、以.tar 结尾的为归档文件，用命令：tar -xvf filename 来展开，生成的文件与源文件在同一目录中，只是少了.tar。

3、以.tar.gz 结尾的文件最常见，可直接用命令：gzip -cd filename | tar xfv 来解开。

tar 的用法：

解压：x 参数表示解压

tar xzf HHARM740.tgz

把一个目录 HHARM740 压缩成一个文件：HHARM740.tgz

tar czf HHARM740.tgz HHARM740

c 参数表示压缩。

9) 查找文件，如：main.c：

find –name main.c

或者：

locate shadow

注意：locate 为模糊匹配，它会递归的在当前目录下的所有子目录下搜索，并列出所有名字包含 shadow 字串的文件。

10) 在一个目录下（含子目录）的所有文件中查找含有某个字符串（如“Modified by hhcn”）的所有文件：

    grep ‘Modified by hhcn’ * -r

11) vi(m)用法

    vi 是 Linux/Unix 世界里极为普遍的全屏幕文本编辑器，几乎可以说任何一台 Linux/Unix 机器都会提供这个软件。

    vi 有三种状态，即编辑方式、插入方式和命令方式。

    - 在命令方式下，所有命令都要以:开始，所键入的字符系统均作命令来处理，如:q 代表退出，:w 表示存盘。
    - 当你进入 vi 时，会首先进入命令方式（同时也是编辑方式）。按下 i 就进入插入方式，用户输入的可视字符都添加到文件中，显示在屏幕上。按下 ESC 就可以回到命令状态（同时也是编辑方式）。
    - 编辑方式和命令方式类似，都是要输入命令，但它的命令不要以:开始，它直接接受键盘输入的单字符或组合字符命令，例如直接按下 u 就表示取消上一次对文件的修改,相当于 WINDOWS 下的 Undo 操作。编译方式下有一些命令是要以/开始的，例如查找字符串就是：/string 则在文件中匹配查找 string 字符串。在编辑模式下按下:就进入命令方式。

基本命令解释：

1. 光标命令

k、j、h、l——上、下、左、右光标移动命令。虽然您可以在 Linux 中使用键盘右边的 4 个光标键，但是记住这 4 个命令还是非常有用的。这 4 个键正是右手在键盘上放置的基本位置。

nG——跳转命令。n 为行数，该命令立即使光标跳到指定行。

Ctrl+G——光标所在位置的行数和列数报告。

w、b——使光标向前或向后跳过一个单词。

2．编辑命令

i、a、r——在光标的前、后以及所在处插入字符命令(i=insert、a=append、r=replace)。

cw、dw——改变(置换)/删除光标所在处的单词的命令 (c=change、d=delete)。

x、d$、dd——删除一个字符、删除光标所在处到行尾的所有字符以及删除整行的命令。

3．查找命令

---- /string、?string——从光标所在处向后或向前查找相应的字符串的命令。

4．拷贝复制命令

---- yy、p——拷贝一行到剪贴板或取出剪贴板中内容的命令。

常用操作：

无论是开启新档或修改旧文件，都可以使用 vi，所需指令为：

$ vi filemane

如果文件是新的，就会在荧幕底部看到一个信息，告诉用户正在创建新文件。如果文件早已存在，vi 则会显示文件的首廿四行，　　　用户可再用光标（cursor）上下移动。

～

～

上面是一个经 vi 开启的模拟文件，一行开始处的波折号（～）表示文件的结尾。

—指令 i 在光标处插入正文

—指令 I 在一行开始处插入正文

—指令 a 在光标後追加正文

—指令 A 在行尾追加正文

—指令 o 在光标下面新开一行

—指令 O 在光标上面新开一行

在插入方式下，不能打入指令，必需先按〈Esc〉键，返回命令方式。假若户不知身处何态，也可以按〈Esc〉键，不管处於何态，都会返回命令方式。

在修改文件时，如何存档及退出指定文件都非常重要。在 vi 内，行使存档或退出的指令时，要先按冒号（:），改变为命令方式，用户就可以看见在荧幕左下方，出现冒号（:），显示 vi 已经改为指　令态，可以进行存档或退出等工作。

:q!放弃任何改动而退出 vi，也就是强行退出

:w 存档

:w! 对于只读文件强行存档

:wq 存档并退出 vi

:x 与 wq 的工作一样

:zz 与 wq 的工作一样删除正文

删除或修改正文都是利用编辑方式，故此，下面所提及的指令只需在编辑方式下，直接键入指令即行。

—x 删除光标处字符（Character）

—nx 删除光标处後 n 个字符

—nX 删除光标处前 n 个字符

—ndw 删除光标处下 n 个单词（word）

—dd 删除整行

—d$或 D 删除由光标至该行最末

—u 恢复前一次所做的删除

当使用 vi 修改正文，加减字符时，就会采用另一组在编辑方式下操作的指令。

— r char 由 char 代替光标处的字符

—Rtext〈Esc〉由 text 代替光标处的字符

—cwtext〈Esc〉由 text 取代光标处的单词

—Ctext〈Esc〉由 text 取代光标处至该行结尾处

—cc 使整行空白，但保留光标位置，让你开始打入

—如删除指令一样，在指令前打入的数，表示执行该指令多少次。

要检索文件，必需在编辑方式下进行。

—／str〈Return〉向前搜寻 str 直至文件结尾处

—?str〈Return〉往後搜寻 str 直至文件开首处

—n 同一方向上重复检索

—N 相反方向上重复检索

—vi 缠绕整个文件，不断检索，直至找到与模式相匹配的下一个出现。

全程替换命令：

:%s/string1/string2/g 在整个文件中替换“string1”成“string2”。

如果要替换文件中的路径：

使用命令“:%s#/usr/bin#/bin#g”可以把文件中所有路径/usr/bin 换成/bin。也可以使用命令“:%s/\/usr\/bin/\/bin/g”实现，其中“\”是转义字符，表明其后的“/”字符是具有实际意义的字符，不是分隔符。

同时编辑 2 个文件，拷贝一个文件中的文本并粘贴到另一个文件中：

命令如下：

---- vi file1 file2

---- yy 在文件 1 的光标处拷贝所在行

---- :n 切换到文件 2 (n=next) 或者按 ctrl+ww，就在两个文件间切换。

---- p 在文件 2 的光标所在处粘贴所拷贝的行

---- :n 切换回文件 1

将文件中的某一部分修改保存到临时文件，例如仅仅把第 20～59 行之间的内容存盘成文件/tmp/1，我们可以键入如下命令。

---- vi file

---- :20,59w /tmp/1

如果要在 vi 执行期间，转到 shell 执行，使用惊叹号（!）执行系统指令，例如在 vi 期间，列出当前目录内容，可以键入 ：

:!ls

另一方面，用户可以在主目录中创建.exrc 环境文件，用 set 打入选项，每次调用 vi 时，就会读入.exrc 中的指令与设置。下面是.exrc 环境文件的实例：

set wrapmarging＝8

set showmode

set autoindent

minicom 用法

minicom 是安装 REDHAT 时安装的软件，它使用配置文件/etc/minirc.dfl，华恒光盘安装时会提供这个文件。

【注意】

minicom 占用串口，能且仅能启动一个 minicom，启动第二个时就会报错：**Device /dev/modem is locked**。其中/dev/modem 就是/dev/ttyS0，即 PC 机串口 1,它是在光盘安装时执行./ucinst 时创建的链接。查看 ucinst 文件，可以看到如下一行：

ln -sf /dev/ttyS0 /dev/modem

minicom 所有的操作都以 ctrl+A 开始，例如：退出为 ctrl+A，松手后再按下 Q，则弹出如下一个小框：选 Yes 即可退出 minicom。

```
Leave without reset?
       Yes          No
```

minicom 中最重要的操作就是对其进行配置的修改。这个操作要先 ctrl+A，松手后按下 O，则弹出如下框：

```
[configuration]
 Filenames and paths
 File transfer protocols
 Serial port setup
 Modem and dialing
 Screen and keyboard
 Save setup as dfl
 Save setup as..
 Exit
```

选择第三项“Serial port setup”，则弹出下面框：

```
A -     Serial Device       : /dev/modem
B - Lockfile Location       : /var/lock
C -    Callin Program       :
D -   Callout Program       :
E -     Bps/Par/Bits        : 19200 8N1
F - Hardware Flow Control   : No
G - Software Flow Control   : No

   Change which setting?
```

键入 E 则弹出如下框，可改变波特率：

```
───────────[Comm Parameters]───────────

 Current: 19200 8N1

   Speed             Parity          Data

 A: 300              L: None         S: 5
 B: 1200             M: Even         T: 6
 C: 2400             N: Odd          U: 7
 D: 4800             O: Mark         V: 8
 E: 9600             P: Space
 F: 19200
 G: 38400
 H: 57600
 I: 115200           Q: 8-N-1
 J: 230400           R: 7-E-1


 Choice, or <Enter> to exit?
```

若要使用 PC 机的串口 2 来接板子的串口 1 做监控，则要在串口配置框中选择 A，即“Serial Device”，则原来的配置框第一行进入编辑模式，将原来的/dev/modem 改为如下的：/dev/ttyS1，即串口 2。

```
A -    Serial Device      : /dev/ttyS1
B - Lockfile Location     : /var/lock
C -   Callin Program      :
D -  Callout Program      :
E -    Bps/Par/Bits       : 19200 8N1
F - Hardware Flow Control : No
G - Software Flow Control : No

   Change which setting?
```

退出配置框只需连续按 ESC 键即可返回。

软、硬盘及光驱的使用

在 Linux 中对其他硬盘逻辑分区、软盘，光盘的使用与我们通常在 DOS 与 Windows 中的使用方法是不一样的，不能直接访问，因为在 Linux 中它们都被视为文件，因此在访问使用前必须使用装载命令 mount 将它们装载到系统的/mnt 目录中来，使用结束，必须进行卸载。命令格式如下：

mount –t 文件系统类型 设备名 装载目录

文件类型常用的有：

msdos　　dos 分区文件
ext2　　Linux 的文件系统
swap　　Linux swap 分区或 swap 文件
iso9660　　安装 CD-ROM 的文件系统
vfat　　支持长文件名的 dos 分区
hpfs　　OS/2 分区文件系统

设备名是指要装载的设备的名称，如软盘、硬盘、光盘等，软盘一般为/dev/fd0 fd1，硬盘一般为/dev/hda hdb ，硬盘逻辑分区一般为期 hda1 hda2…等等，光盘一般为/dev/hdc。在装载前一般要在/dev/mnt 目录下建立一个空的目录，如软盘为 floppy，硬盘分区为其盘符如 c、d 等等，光

盘为 cd-rom，使用命令：

mount –t msdos /dev/fd0 /mnt/floppy

装载一个 mddos 格式的软盘

mount –t ext2 /dev/fd0 /mnt/floppy

装载一个 Linux 格式的软盘

mount –t vfat /dev/hda1 /mnt/c

装载 Windows98 格式的硬盘分区

mount –t iso9660 /dev/hdc /mnt/cd-rom

装载一个光盘

装载完成之后便可对该目录进行操作，在使用新的软盘及光盘前必须退出该目录，使用卸载命令进行卸载，方可使用新的软盘及光盘，否则系统不会承认该软盘的，光盘在卸载前是不能用光驱面板前的弹出键退出的。

LILO 与 GRUB

LINUX 安装时一般都安装 bootloader，可以支持多操作系统并存。常见的 bootloader 有 LILO 和 GRUB，REDHAT6.x 使用 LILO，REDHAT7.2 同时支持 LILO 和 GRUB，但默认使用的是 GRUB。

Diff 创建软件补丁，用 patch 打补丁

diff 是生成源码补丁的必备工具。其命令格式为：

diff [命令行选项] 原始文件 新文件

常用命令行选项如下：

-r 递归处理目录　　　-u 输出统一格式(unified format)

-N patch 里包含新文件　　-a patch 里可以包含二进制文件

它的输出在 stdout 上，所以你可能需要把它重定向到一个文件。

输出格式保存了上下文（缺省是上下各三行，最少需要两行），这样，patch 的时候可以允许行号不精确匹配的情况出现。另外，在 patch 文件的开头明确地用---和+++标示出原始文件和当前文件，也方便阅读。

通常，我们需要对整个软件包做修改，并生成一个 patch 文件，下面是典型的操作过程。

```
tar xzvf software.tar.gz # 展开原始软件包，其目录为 software
cp _a software software-orig # 做个修改前的备份
cd software
[修改，测试……]
cd ..
diff -ruNa software-orig software > software-my.patch
```

现在我们就可以保存 software-my.patch 做为这次修改的结果，至于原始软件包，可以不必保存。等到下次需要再修改的时候，可以用 patch 命令把这个补丁打进原始包，再继续工作。比如是在 linux kernel 上做的工作，就不必每次保存几十兆修改后的源码了。这是好处之一，好处之二是维护方便，由于 unified patch 格式有一定的模糊匹配能力，能减少原软件包升级带来的维护工作量。

patch

patch 程序根据补丁(patch)文件修改一个文件。补丁文件通常是使用 diff 程序建立的一个列表，这个列表包含如何修改原始文件的一些指令。由于节省时间和空间，Patch 经常用于源代码的修补。可以想象一个有 1MB 的程序包，这个程序包的下一个版本仅仅改变了前面一个版本的两个文件的内容，这个新版本可以完全以一个 1MB 的新版本进行发布或者以一个仅仅有 1KB 的补丁文件进行发布。这个补丁文件可以对第一个版本的进行更新，更新后的版本就和第二个版本完全一致了。因此，如果已经下载了第一个版本，那么为了下一个版本而进行的大数据量下载工作就可以有效地避免。

常用命令行选项：

patch [命令行选项] [待 patch 的文件[patch]]

-pn patch level(n 是数字) -b[后缀] 生成备份，缺省是.orig

为了说明什么是 patch level，这里看一个 patch 文件的头标记。

```
diff -ruNa xc.orig/config/cf/Imake.cf xc.bsd/config/cf/Imake.cf
```

```
--- xc.orig/config/cf/Imake.cf Fri Jul 30 12:45:47 1999
+++ xc.new/config/cf/Imake.cf Fri Jan 21 13:48:44 2000
```

这个 patch 如果直接应用,它会去找 xc.orig/config/cf 目录下的 Imake.cf 文件，假如你的源码树的根目录是缺省的 xc 而不是 xc.orig，除了 mv xc xc.orig 之外，有无简单的方法应用此 patch 呢？patch level 就是为此而设：patch 会把目标路径名砍去开头 patch level 个节(由/分开的部分)。在本例中，可以用下述命令：

```
cd xc; patch -p1 < /pathname/xxx.patch
```

完成操作。注意，由于没有指定 patch 文件，patch 程序默认从 stdin 读入，所以用了输入重定向。

又例如：

```
diff -r dir1 dir2 >patch20020523.patch
```

递归的比较目录 dir1 与 dir2 内，所有各文件之不同处，并将不同处记录到 patch20020523.patch 文件中。

```
patch -p1 < [patchfile]
```

-p1 选项代表 patchfile 中文件名左边目录的层数，顶层目录在不同的机器上有所不同。要使用这个选项，就要把你的 patch 放在要被打补丁的目录下，然后在这个目录中运行 path -p1 < [patchfile]。

LINUX 下的硬盘分区

对习惯于使用 Dos 或 Windows 的用户来说，有几个分区就有几个驱动器，并且每个分区都会获得一个字母标识符，然后就可以选用这个字母来指定在这个分区上的文件和目录，它们的文件结构都是独立的，非常好理解。但对 Red Hat Linux 用户来说无论有几个分区，分给哪一目录使用，它归根结底就只有一个根目录，一个独立且唯一的文件结构。Red Hat Linux 中每个分区都是用来组成整个文件系统的一部分，因为它采用了一种叫“ 载入 ” 的处理方法，它的整个文件系统中包含了一整套的文件和目录，且将

一个分区和一个目录联系起来。这时要载入的一个分区将使它的存储空间在一个目录下获得。

对于 IDE 硬盘，驱动器标识符为“hdx~”,其中“hd”表明分区所在设备的类型，这里是指 IDE 硬盘了。“x”为盘号（a 为基本盘，b 为基本从属盘，c 为辅助主盘，d 为辅助从属盘）,“~”代表分区，前四个分区用数字 1 到 4 表示，它们是主分区或扩展分区，从 5 开始就是逻辑分区。例，hda3 表示为第一个 IDE 硬盘上的第三个主分区或扩展分区,hdb2 表示为第二个 IDE 硬盘上的第二个主分区或扩展分区。对于 SCSI 硬盘则标识为“sdx~”，SCSI 硬盘是用“sd”来表示分区所在设备的类型的，其余则和 IDE 硬盘的表示方法一样。

从上面可以看到，Red Hat Linux 的分区是不同于其它操作系统分区的，它的分区格式只有 Ext2（3）和 Swap 两种，Ext2（3）用于存放系统文件，Swap 则作为 Red Hat Linux 的交换分区。Red Hat Linux 至少需要两个专门的分区（Linux Native 和 Linux Swap）况且不能将 Red Hat Linux 安装在 Dos/Windows 分区。一般来说将 Red Hat Linux 安装一个或多个类型为“Linux Native”的硬盘分区，但是在 Red Hat Linux 的每一个分区都必须要指定一个“Mount Point”（载入点），告诉 Red Hat Linux 在启动时，这个目录要给哪个目录使用。对“Swap”分区来说,一般定义一个且它不必要定义载入点。

SWAP 分区是 LINUX 暂时存储数据的交换分区，它主要是把主内存上暂时不用的数据存起来，在需要的时候再调进内存内，且作为 SWAP 使用的分区不用指定“Mout Point”（载入点），既然它作为交换分区，我们理所当然应给它指定大小，它至少要等于系统上实际内存的量，一般来说它的大小是内存的两倍，如果你是 16MB 的内存，那么 SWAP 分区的大小是 32MB 左右，以此类推。但必须还要注意一点，SWAP 分区不要大于 128MB，如果你是 64MB 的内存，那么 SWAP 分区最大也只能被定为 127MB，再大就是浪费空间了，因为系统不需要太大的交换分区。以此类推，如果你是 128MB 或更大的内存，SWAP 分区也只能最大被定为 127MB。可以创建和使用一个以

上的交换分区，最多 16 个。

Linux Native 是存放系统文件的地方，它只能用 EXT2（3）的分区类型。对 Windows 用户来说，操作系统必须装在同一分区里，它是商业软件吗!所以你没有选择的余地！对 Red Hat Linux 来说，你有了较大的选择余地，你可以把系统文件分几个区来装（必须要说明载入点），也可以就装在同一个分区中（载入点是“ /”）。

/boot 分区，它包含了操作系统的内核和在启动系统过程中所要用到的文件，建这个分区是有必要的，因为目前大多数的 PC 机要受到 BIOS 的限制，况且如果有了一个单独的/boot 启动分区，即使主要的根分区出现了问题，计算机依然能够启动。这个分区的大小约在 50MB—100MB 之间。但是如果想用 LILO 启动 Red Hat Linux 系统的话，含有/boot 的分区必须完全在柱面 1023 以下。又由于 8GB 后的数据 LILO 不能读取，所以 Red Hat Linux 要安装在 8GB 的区域以内。

/usr 分区，是 Red Hat Linux 系统存放软件的地方，如有可能应将最大空间分给它。

/home 分区，是用户的 home 目录所在地，这个分区的大小取决于有多少用户。如果是多用户共同使用一台电脑的话，这个分区是完全有必要的，况且根用户也可以很好地控制普通用户使用计算机，如对用户或者用户组实行硬盘限量使用，限制普通用户访问哪些文件等。其实单用户也有建立这个分区的必要，因为没这个分区的话，那么你只能以 root 用户的身份登陆系统，这样做是危险的，因为 root 用户对系统有绝对的使用权，一旦对系统进行了误操作，就会导致系统崩溃。

/var/log 分区，是系统日志记录分区，如果设立了这一单独的分区，这样即使系统的日志文件出现了问题，它们也不会影响到操作系统的主分区。

/tmp 分区，用来存放临时文件。这对于多用户系统或者网络服务器来说是有必要的。这样即使程序运行时生成大量的临时文件，或者用户对系统进行了错误的操作，文件系统的其它部分仍然是安全的。因为文件系统的这一部分仍然还承受着读写操作，所以它通常会比其它的部分更快地发生问

题。
/bin 分区，存放标准系统实用程序。
/dev 分区，存放设备文件。
/opt 分区，存放可选的安装的软件。
/sbin 分区，存放标准系统管理可执行文件，如 insmod，ifconfig 等。
上面介绍了几个常用的分区，一般来说需要一个 SWAP 分区，一个/boot 分区，一个/usr 分区，一个/home 分区，一个/var/log 分区。当然这没有什么规定，完全是依照个人来定的。但记住至少要有两个分区，一个 SWAP 分区，一个/分区。
用户可以使用两种分区工具：
1. Disk Druid：它是 Red Hat Linux 提供的硬盘管理工具，它最初是随 Red HatLinux5 一起发售的，它可以根据用户的要求创建和删除硬盘分区，另外还可以为每个分区管理载入点，这是一个不错的分区软件，建议读者使用。本文也将以此软件详细地介绍 Red Hat Linux 分区。
2. Fdisk：它是传统的 Linux 硬盘分区工具，比 Disk Druid 更强大，使用更加灵活。但是 Fdisk 要求用户对硬盘分区有一定经验，并能够适应且读懂简单的文本界面。如果你是第一次对一个硬盘驱动器进行分区操作的话，最好还是避免 Fdisk 这样的程序，它虽然强大但用起来的感觉不是太好的。

附录 C gcc 与 gdb

gcc 是 GNU 的 C 和 C++ 编译器，它是 Linux 中最重要的软件开发工具。实际上，gcc 能够编译三种语言：C、C++ 和 Object C（C 语言的一种面向对象扩展）。利用 gcc 命令可同时编译并连接 C 和 C++ 源程序。汇编语言的编译器为 as。

编译器被成功的移植到不同的处理器平台上。标准 PC LINUX 上的 gcc 是 FOR INTEL CPU 的，而华恒 ARM 系列开发套件使用的是 FOR armnommu 系列处理器的 gcc 编译器 arm-elf-gcc 和 arm-elf-as 及其相应的 GNU Binutils 工具集（如 ld 链接工具，objcopy、objdump 等工具）。

gcc 命令的常用选项有：

-ansi 只支持 ANSI 标准的 C 语法。这一选项将禁止 GNU C 的某些特色，

例如 asm 或 typeof 关键词。

-c 只编译并生成目标文件。

-DMACRO 以字符串“1”定义 MACRO 宏。

-DMACRO=DEFN 以字符串“DEFN”定义 MACRO 宏。

-E 只运行 C 预编译器。

-g 生成调试信息。GNU 调试器可利用该信息。

-IDIRECTORY 指定额外的头文件搜索路径 DIRECTORY。

-LDIRECTORY 指定额外的函数库搜索路径 DIRECTORY。

-lLIBRARY 连接时搜索指定的函数库 LIBRARY。

-m486 针对 486 进行代码优化。

-o FILE 生成指定的输出文件。用在生成可执行文件时。

-O0 不进行优化处理。

-O 或 -O1	优化生成代码。
-O2	进一步优化。
-O3	比 -O2 更进一步优化，包括 inline 函数。
-shared	生成共享目标文件。通常用在建立共享库时。
-static	禁止使用共享连接。
-UMACRO	取消对 MACRO 宏的定义。
-w	不生成任何警告信息。
-Wall	生成所有警告信息。

ld 文件

编译完成之后，就要执行 ld 进行链接。ld 工具处理 ld 文件。

ld 文件采用 AT&T 链接命令语言写成,用于控制整个链接过程。

GDB

Linux 包含了一个叫 gdb 的 GNU 调试程序. gdb 是一个用来调试 C 和 C++程序的强力调试器. 它使你能在程序运行时观察程序的内部结构和内存的使用情况. Gdb 功能非常强大:

可监视程序中变量的值.

可设置断点以使程序在指定的代码行上停止执行.

支持单步执行等

在命令行上键入 gdb 并按回车键就可以运行 gdb 了, 如果一切正常的话, gdb 将被启动并且你将在屏幕上看到类似的内容:

```
GNU gdb Red Hat Linux 7.x (5.0rh-15) (MI_OUT)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and
you are
welcome to change it and/or distribute copies of it under certain
```

```
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "i386-redhat-linux".
(gdb)
```

当你启动 gdb 后，你能在命令行上指定很多的选项．你也可以以下面的方式来运行 gdb :

gdb <fname>

当你用这种方式运行 gdb ，你能直接指定想要调试的程序．这将告诉 gdb 装入名为 fname 的可执行文件．你也可以用 gdb 去检查一个因程序异常终止而产生的 core 文件，或者与一个正在运行的程序相连．你可以参考 gdb 指南页或在命令行上键入 gdb -h 得到一个有关这些选项的说明的简单列表．

为了使 gdb 正常工作，你必须使你的程序在编译时包含调试信息．调试信息包含你程序里的每个变量的类型和在可执行文件里的地址映射以及源代码的行号． gdb 利用这些信息使源代码和机器码相关联．

在编译时用 -g 选项打开调试选项．

gdb 支持很多的命令使你能实现不同的功能．这些命令从简单的文件装入到允许你检查所调用的堆栈内容的复杂命令，表 27.1 列出了你在用 gdb 调试时会用到的一些命令．想了解 gdb 的详细使用请参考 gdb 的指南页．

gdb 的常用命令

break NUM　　　　　　在指定的行上设置断点。

bt　　　　　　　　　显示所有的调用栈帧。该命令可用来显示函数的调用顺序。

clear　　　　　　　删除设置在特定源文件、特定行上的断点。其用法为：clear FILENAME:NUM。

continue　　　　　　继续执行正在调试的程序。该命令用在程序由

于处理信号或断点而

导致停止运行时。

display EXPR 每次程序停止后显示表达式的值。表达式由程序定义的变量组成。

file FILE 装载指定的可执行文件进行调试。

help NAME 显示指定命令的帮助信息。

info break 显示当前断点清单，包括到达断点处的次数等。

info files 显示被调试文件的详细信息。

info func 显示所有的函数名称。

info local 显示当函数中的局部变量信息。

info prog 显示被调试程序的执行状态。

info var 显示所有的全局和静态变量名称。

kill 终止正被调试的程序。

list 显示源代码段。

make 在不退出 gdb 的情况下运行 make 工具。

next 在不单步执行进入其他函数的情况下，向前执行一行源代码。

print EXPR 显示表达式 EXPR 的值。

gdb 支持很多与 UNIX shell 程序一样的命令编辑特征．你能象在 bash 或 tcsh 里那样按 Tab 键让 gdb 帮你补齐一个唯一的命令，如果不唯一的话 gdb 会列出所有匹配的命令．你也能用光标键上下翻动历史命令．

gdb 应用举例

下面用一个实例教你一步步的用 gdb 调试程序．被调试的程序相当的简单，但它展示了 gdb 的典型应用．

下面列出了将被调试的程序．这个程序被称为 greeting ，它显示一个简单的问候，再用反序将它列出．

```
#include  <stdio.h>
```

```
main (){
  char my_string[] = "hello there";
  my_print (my_string);
  my_print2 (my_string);
}
void my_print (char *string){
  printf ("The string is %s\n", string);
}
void my_print2 (char *string){
  char *string2;
  int size, i;
  size = strlen (string);
  string2 = (char *) malloc (size + 1);
  for (i = 0; i < size; i++)
    string2[size - i] = string[i];
  string2[size+1] = `\0';
  printf ("The string printed backward is %s\n", string2);
}
```

用下面的命令编译它:

```
gcc -o test test.c
```

这个程序执行时显示如下结果:

```
The string is hello there
The string printed backward is
```

输出的第一行是正确的, 但第二行打印出的东西并不是我们所期望的. 我们所设想的输出应该是:

```
The string printed backward is ereht olleh
```

由于某些原因, my_print2 函数没有正常工作. 让我们用 gdb 看看问题究竟出在哪儿, 先键入如下命令:

```
gdb greeting
```

【注意】

记得在编译 greeting 程序时把调试选项打开.

如果你在输入命令时忘了把要调试的程序作为参数传给 gdb ，你可以在 gdb 提示符下用 file 命令来载入它:

```
(gdb) file greeting
```

这个命令将载入 greeting 可执行文件就象你在 gdb 命令行里装入它一样. 这时你能用 gdb 的 run 命令来运行 greeting 了. 当它在 gdb 里被运行后结果大约会象这样:

```
(gdb) run
Starting program: /root/greeting
The string is hello there
The string printed backward is
Program exited with code 041
```

这个输出和在 gdb 外面运行的结果一样. 问题是, 为什么反序打印没有工作? 为了找出症结所在, 我们可以在 my_print2 函数的 for 语句后设一个断点, 具体的做法是在 gdb 提示符下键入 list 命令三次, 列出源代码:

```
(gdb) list
(gdb) 回车
(gdb) 回车
```

（在 gdb 提示符下按回车健将重复上一个命令。）

要 list 三次是因为一次无法显示全部文件内容，而必须多次才能翻到想要设置断点的文件行处。根据列出的源程序, 能看到要设断点的地方在第 24 行, 在 gdb 命令行提示符下键入如下命令设置断点:

```
(gdb) break 24
```

gdb 将作出如下的响应:

```
Breakpoint 1 at 0x139: file greeting.c, line 24
```

(gdb)

现在再键入 run 命令，将产生如下的输出:

```
Starting program: /root/greeting
The string is hello there
Breakpoint 1, my_print2 (string = 0xbfffdc4 "hello there") at greeting.c :24
24  string2[size-i]=string[i]
```

通过设置一个观察 string2[size - i] 变量的值的观察点来看出错误是怎样产生的，做法是键入:

```
(gdb) watch string2[size - i]
```

gdb 将作出如下回应:

```
Watchpoint 2: string2[size - i]
```

现在可以用 next 命令来一步步的执行 for 循环了:

```
(gdb) next
```

经过第一次循环后， gdb 告诉我们 string2[size - i] 的值是 `h`. gdb 用如下的显示来告诉你这个信息:

```
Watchpoint 2, string2[size - i]
Old value = 0 `\000'
New value = 104 `h'
my_print2(string = 0xbfffdc4 "hello there") at greeting.c:23
23 for (i=0; i<size; i++)
```

这个值正是期望的. 后来的数次循环的结果都是正确的. 当 i=10 时，表达式 string2[size - i] 的值等于 `e`， size - i 的值等于 1，最后一个字符已经拷到新串里了.

如果你再把循环执行下去，你会看到已经没有值分配给 string2[0] 了， 而它是新串的第一个字符，因为 malloc 函数在分配内存时把它们初始化为空(null)字符. 所以 string2 的第一个字符是空字符. 这解释了为什么在打印 string2 时没有任何输出了.

现在找出了问题出在哪里, 修正这个错误是很容易的. 你得把代码里写入 string2 的第一个字符的的偏移量改为 size - 1 而不是 size. 这是因为 string2 的大小为 12, 但起始偏移量是 0, 串内的字符从偏移量 0 到偏移量 10, 偏移量 11 为空字符保留.

附录 D Makefile

GNU make

在大型的开发项目中，通常有几十到上百个的源文件，如果每次均手工键入 gcc 命令进行编译的话，则会非常不方便。因此，人们通常利用 make 工具来自动完成编译工作。这些工作包括：如果仅修改了某几个源文件，则只重新编译这几个源文件；如果某个头文件被修改了，则重新编译所有包含该头文件的源文件。利用这种自动编译可大大简化开发工作，避免不必要的重新编译。

实际上，make 工具通过一个称为 makefile 的文件来完成并自动维护编译工作。makefile 需要按照某种语法进行编写，其中说明了如何编译各个源文件并连接生成可执行文件，并定义了源文件之间的依赖关系。

当修改了其中某个源文件时，如果其他源文件依赖于该文件，则也要重新编译所有依赖该文件的源文件。

makefile 文件是许多编译器，包括 Windows NT 下的编译器维护编译信息的常用方法，只是在集成开发环境中，用户通过友好的界面修改 makefile 文件而已。默认情况下，GNU make 工具在当前工作目录中按如下顺序搜索 makefile：

* GNUmakefile
* makefile
* Makefile

在 UNIX 系统中，习惯使用 Makefile 作为 makfile 文件。如果要使用其他文件作为 makefile，则可利用类似下面的 make 命令选项指定 makefile 文件：

```
$ make -f Makefile.debug
```

makefile 基本结构

makefile 中一般包含如下内容：

* 需要由 make 工具创建的项目，通常是目标文件和可执行文件。通常使用“目标（target）”一词来表示要创建的项目。
* 要创建的项目依赖于哪些文件。
* 创建每个项目时需要运行的命令。

例如，假设你现在有一个 C++ 源文件 test.C，该源文件包含有自定义的头文件 test.h，则目标文件 test.o 明确依赖于两个源文件：test.c 和 test.h。另外，你可能只希望利用 g++ 命令来生成 test.o 目标文件。这时，就可以利用如下的 makefile 来定义 test.o 的创建规则：

```
# This makefile just is a example.
# The following lines indicate how test.o depends
# test.C and test.h, and how to create test.o

test.o: test.c test.h
    g++ -c -g test.C
```

从上面的例子注意到，第一个字符为 # 的行为注释行。第一个非注释行指定 test.o 为目标，并且依赖于 test.c 和 test.h 文件。随后的行指定了如何从目标所依赖的文件建立目标。当 test.c 或 test.h 文件在编译之后又被修改，则 make 工具可自动重新编译 test.o，如果在前后两次编译之间，test.C 和 test.h 均没有被修改，而且 test.o 还存在的话，就没有必要重新编译。这种依赖关系在多源文件的程序编译中尤其重要。通过这种依赖关系的定义，make 工具可避免许多不必要的编译工作。当然，利用 Shell 脚本也可以达到自动编译的效果，但是，Shell 脚本将全部编译任何源文件，包括哪些不必要重新编译的源文件，而 make 工具则可根据目标上一次编译的时间和目标所依赖的源文件的更新时间而自动判断应当编译哪个源文件。

一个 makefile 文件中可定义多个目标，利用 make target 命令可指定要编译的目标，如果不指定目标，
则使用第一个目标。通常，makefile 中定义有 clean 目标，可用来清除编译过程中的中间文件，例如：

```
clean:
    rm -f *.o
```

运行 make clean 时，将执行 rm -f *.o 命令，最终删除所有编译过程中产生的所有中间文件。

makefile 变量

GNU 的 make 工具除提供有建立目标的基本功能之外，还有许多便于表达依赖性关系以及建立目标的命令的特色。其中之一就是变量或宏的定义能力。如果你要以相同的编译选项同时编译十几个 C 源文件，而为每个目标的编译指定冗长的编译选项的话，将是非常乏味的。但利用简单的变量定义，可避免这种乏味的工作：

```
# Define macros for name of compiler
CC = gcc

# Define a macr o for the CC flags
CCFLAGS = -D_DEBUG -g -m486

# A rule for building a object file
test.o: test.c test.h
    $(CC) -c $(CCFLAGS) test.c
```

在上面的例子中，CC 和 CCFLAGS 就是 make 的变量。GNU make 通常称之为变量，而其他 UNIX 的 make 工具称之为宏，实际是同一个东西。在 makefile 中引用变量的值时，只需变量名之前添加 $ 符号，如上面的 $(CC)

和 $(CCFLAGS)。

GNU make 的主要预定义变量 i

GNU make 有许多预定义的变量，这些变量具有特殊的含义，可在规则中使用。表 1-5 给出了一些主要的
预定义变量，除这些变量外，GNU make 还将所有的环境变量作为自己的预定义变量。

$* 不包含扩展名的目标文件名称。

$+ 所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件。

$< 第一个依赖文件的名称。

$? 所有的依赖文件，以空格分开，这些依赖文件的修改日期比目标的创建日期晚。

$@ 目标的完整名称。

$^ 所有的依赖文件，以空格分开，不包含重复的依赖文件。

$% 如果目标是归档成员，则该变量表示目标的归档成员名称。例如，如果目标名称为 mytarget.so(image.o)，则 $@ 为 mytarget.so，而 $% 为 image.o。

AR 归档维护程序的名称，默认值为 ar。

ARFLAGS 归档维护程序的选项。

AS 汇编程序的名称，默认值为 as。

ASFLAGS 汇编程序的选项。

CC C 编译器的名称，默认值为 cc。

CCFLAGS C 编译器的选项。

CPP C 预编译器的名称，默认值为 $(CC) -E。

CPPFLAGS C 预编译的选项。

CXX C++ 编译器的名称，默认值为 g++。

CXXFLAGS C++ 编译器的选项。

FC　　　　　　FORTRAN 编译器的名称，默认值为 f77。
FFLAGS　　　　FORTRAN 编译器的选项。

隐含规则

GNU make 包含有一些内置的或隐含的规则，这些规则定义了如何从不同的依赖文件建立特定类型的目标。

GNU make 支持两种类型的隐含规则：

* 后缀规则（Suffix Rule）。后缀规则是定义隐含规则的老风格方法。后缀规则定义了将一个具有某个后缀的文件（例如，.c 文件）转换为具有另外一种后缀的文件（例如，.o 文件）的方法。每个后缀规则以两个成对出现的后缀名定义，例如，将 .c 文件转换为 .o 文件的后缀规则可定义为：

```
.c.o:
$(CC) $(CCFLAGS) $(CPPFLAGS) -c -o $@ $<
```

* 模式规则（pattern rules）。这种规则更加通用，因为可以利用模式规则定义更加复杂的依赖性规则。模式规则看起来非常类似于正则规则，但在目标名称的前面多了一个 % 号，同时可用来定义目标和依赖文件之间的关系，例如下面的模式规则定义了如何将任意一个 X.c 文件转换为 X.o 文件：

```
%.c:%.o
$(CC) $(CCFLAGS) $(CPPFLAGS) -c -o $@ $<
```

makefile 范例

运行 make

我们知道，直接在 make 命令的后面键入目标名可建立指定的目标，如果

直接运行 make，则建立第一个目标。我们还知道可以用 make -f mymakefile 这样的命令指定 make 使用特定的 makefile，而不是默认的 GNUmakefile、makefile 或 Makefile。但 GNU make 命令还有一些其他选项，下面给出了这些选项。

-C DIR　　在读取 makefile 之前改变到指定的目录 DIR。
-f FILE　　以指定的 FILE 文件作为 makefile。
-h　　显示所有的 make 选项。
-i　　忽略所有的命令执行错误。
-I DIR　　当包含其他 makefile 文件时，可利用该选项指定搜索目录。
-n　　只打印要执行的命令，但不执行这些命令。
-p　　显示 make 变量数据库和隐含规则。
-s　　在执行命令时不显示命令。
-w　　在处理 makefile 之前和之后，显示工作目录。
-W FILE　　假定文件 FILE 已经被修改。

附录 E uClinux 系统分析

uClinux 简介

Linux 是一种很受欢迎的操作系统，它与 UNIX 系统兼容，开放源代码。它原本被设计为桌面系统，现在广泛应用于服务器领域。而更大的影响在于它正逐渐的应用于嵌入式系统领域。uClinux 正是在这种氛围下产生的。在 uClinux 这个英文单词中 u 表示 Micro，小的意思，C 表示 Control，控制的意思，所以 uClinux 就是 Micro-Control-Linux，字面上的理解就是“针对微控制领域而设计的 Linux 系统”。

网络资源：

www.uclinux.org

uClinux 小型化的做法

标准 Linux 可能采用的小型化方法

重新编译内核

Linux 内核采用模块化的设计，即很多功能块可以独立的加上或卸下，开发人员在设计内核时把这些内核模块作为可选的选项，可以在编译系统内核时指定。因此一种较通用的做法是对 Linux 内核重新编译，make munuconfig 在编译时仔细的选择嵌入式设备所需要的功能支持模块，同时删除不需要的功能。通过对内核的重新配置，可以使系统运行所需要的内核显著减小，从而缩减资源使用量。

制作 root 文件系统映象

Linux 系统在启动时必须加载根（root）文件系统，因此剪裁系统同时包括 root file system 的剪裁。在 x86 系统下，Linux 可以在 Dos 下，使用 Loadlin 文件加载启动，

uClinux 采用的小型化方法

1．uClinux 的内核加载方式

uClinux 的内核有两种可选的运行方式：可以在 flash 上直接运行，也可以加载到内存中运行。

Flash 运行方式：把内核的可执行映象烧写到 flash 上，系统启动时从 flash 的某个地址开始逐句执行。这种方法实际上是很多嵌入式系统采用的方法。这种做法可以减少内存需要。

内核加载方式：把内核的压缩文件存放在 flash 上，系统启动时读取压缩文件在内存里解压，然后开始执行，这种方式相对复杂一些，但是运行速度可能更快（RAM 的存取速率要比 flash 高）。同时这也是标准 Linux 系统采用的启动方式。

2．uClinux 的根（root）文件系统

uClinux 系统采用 romfs 文件系统，这种文件系统相对于一般的 ext2 文件系统要求更少的空间。空间的节约来自于两个方面，首先内核支持 romfs 文件系统比支持 ext2 文件系统需要更少的代码，其次 romfs 文件系统相对简单，在建立文件系统超级块（superblock）需要更少的存储空间。Romfs 文件系统不支持动态擦写保存，对于系统需要动态保存的数据采用虚拟 ram 盘的方法进行处理（ram 盘将采用 ext2 文件系统）。

3．uClinux 的应用程序库

uClinux 小型化的另一个做法是重写了应用程序库，相对于越来越大且越来越全的 glibc 库，uClibc 对 libc 做了精简。

uClinux 对用户程序采用静态连接的形式，这种做法会使应用程序变大，但是基于内存管理的问题，不得不这样做（这将在下文对 uClinux 内存管理展开分析时进行说明），同时这种做法也更接近于通常嵌入式系统的做法。

uClinux 的开发环境

环境建立

华恒公司 ARM 开发套件附送光盘中安装完成后即建立了完备的 uClinux 开发环境。

GNU 开发套件

Gnu 开发套件作为通用的 Linux 开放套件，包括一系列的开发调试工具。

主要组件：

Gcc： 编译器，可以做成交叉编译的形式，即在宿主机上开发编译目标上可运行的二进制文件。

Binutils：一些辅助工具，包括 objdump（可以反编译二进制文件），as（汇编编译器），ld（连接器），objcopy 等等。

gdb：调试器，可使用多种交叉调试方式，gdb-JTAG（背景调试工具），gdbserver（使用以太网进行远程调试）。

uClinux 的打印终端

通常情况下，uClinux 的默认的标准输入输出被重定向到串口 1（/dev/ttyS0），内核在启动时所有的信息都打印到串口终端，同时也可以通过串口终端与系统交互。uClinux 在启动时启动了 telnetd（远程登录服务），操作者可以远程登录上系统，从而控制系统的运行。至于是否允许远程登录可以通过烧写 romfs 文件系统时有用户决定是否启动远程登录服务。

交叉编译调试工具

支持一种新的处理器，必须具备一些编译，汇编工具，使用这些工具可以形成可运行于这种处理器的二进制文件。对于内核使用的编译工具同应用程序使用的有所不同。在解释不同点之前，需要对 gcc 连接做一些说明：

.ld（link description）文件：ld 文件是指出连接时内存映象格式的文件。

crt0.S：应用程序编译连接时需要的启动文件，主要是初始化应用程序栈。

pic：position independence code ，与位置无关的二进制格式文件，在程序段中必须包括 reloc 段，从而使的代码加载时可以进行重新定位。

内核编译连接时，使用.ld 文件，形成可执行文件映象，所形成的代码段既可以使用间接寻址方式（即使用 reloc 段进行寻址），也可以使用绝对寻址方式。这样可以给编译器更多的优化空间。因为内核可能使用绝对寻址，

所以内核加载到的内存地址空间必须与 ld 文件中给定的内存空间完全相同。

应用程序的连接与内核连接方式不同。应用程序由内核加载（可执行文件加载器将在后面讨论），由于应用程序的 ld 文件给出的内存空间与应用程序实际被加载的内存位置可能不同，这样在应用程序加载的过程中需要一个重新地位的过程，即对 reloc 段进行修正，使得程序进行间接寻址时不至于出错。(这个问题在 i386 等高级处理器上方法有所不同，本文将在后面进一步分析)。

由上述讨论，至少需要两套编译连接工具。在讨论过 uClinux 的内存管理后本文将给出整个系统的工作流程以及系统在 flash 和 ram 中的空间分布。

可执行文件格式

先对一些名词作一些说明：

coff（common object file format）：一种通用的对象文件格式

elf（excutive linked file）：一种为 Linux 系统所采用的通用文件格式，支持动态连接

flat：elf 格式有很大的文件头，flat 文件对文件头和一些段信息做了简化

uClinux 系统只支持 flat 可执行文件格式，gcc 的编译器不能直接形成这种文件格式，但是可以形成 coff 或 elf 格式的可执行文件，这两种文件需要 coff2flt 或 elf2flt 工具进行格式转化，形成 flat 文件。

当用户执行一个应用时，内核的执行文件加载器将对 flat 文件进行进一步处理，主要是对 reloc 段进行修正（可执行文件加载器的详见 fs/binfmt_flat.c)。以下对 reloc 段进一步讨论。

需要 reloc 段的根本原因是，程序在连接时连接器所假定的程序运行空间与实际程序加载到的内存空间不同。假如有这样一条指令：

jsr app_start;

这一条指令采用直接寻址，跳转到 app_start 地址处执行，连接程序将在编译完成是计算出 app_start 的实际地址（设若实际地址为 0x10000)，这个实际地址是根据 ld 文件计算出来（因为连接器假定该程序将被加载到由 ld

文件指明的内存空间）。但实际上由于内存分配的关系，操作系统在加载时无法保证程序将按 ld 文件加载。这时如果程序仍然跳转到绝对地址 0x10000 处执行，通常情况这是不正确的。一个解决办法是增加一个存储空间，用于存储 app_start 的实际地址，设若使用变量 addr 表示这个存储空间。则以上这句程序将改为：

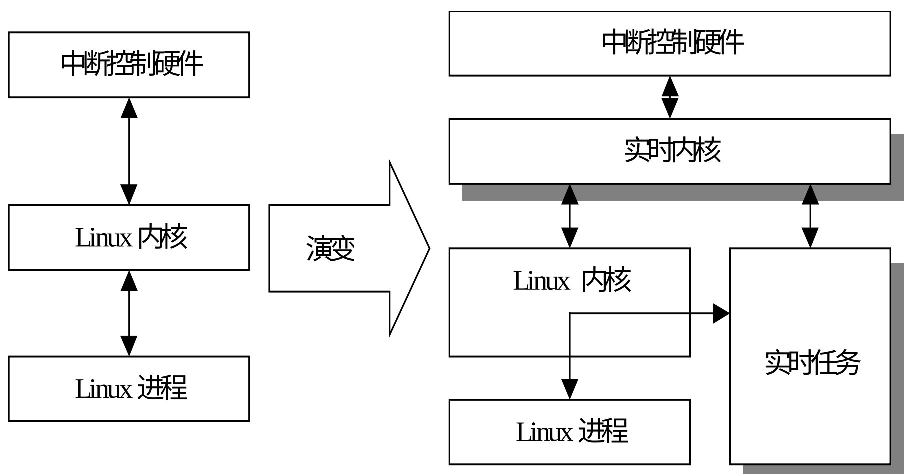
```
movl      addr,      a0;
   jsr(a0);
```

增加的变量 addr 将在数据段中占用一个 4 字节的空间，连接器将 app_start 的绝对地址存储到该变量。在可执行文件加载时，可执行文件加载器根据程序将要加载的内存空间计算出 app_start 在内存中的实际位置，写入 addr 变量。系统在实际处理时是不需要知道这个变量的确切存储位置（也不可能知道），系统只要对整个 reloc 段进行处理就可以了（reloc 段有标识，系统可以读出来）。处理很简单只需要对 reloc 段中存储的值统一加上一个偏置（如果加载的空间比预想的要靠前，实际上是减去一个偏移量）。偏置由实际的物理地址起始值同 ld 文件指定的地址起始值相减计算出。

这种 reloc 的方式部分是由 uClinux 的内存分配问题引起的，这一点将在 uClinux 内存管理分析时说明。

针对实时性的解决方案

uClinux 本身并没有关注实时问题，它并不是为了 Linux 的实时性而提出的。另外有一种 Linux——RTAI-linux 关注实时问题。RTAI-linux 执行管理器把普通 Linux 的内核当成一个任务运行，同时还管理了实时进程。而非实时进程则交给普通 Linux 内核处理。这种方法已经应用于很多的操作系统用于增强操作系统的实时性，包括一些商用版 UNIX 系统，Windows NT 等等。这种方法优点之一是实现简单，且实时性能容易检验。优点之二是由于非实时进程运行于标准 Linux 系统，同其它 Linux 商用版本之间保持了很大的兼容性。优点之三是可以支持硬实时时钟的应用。



uClinux 可以使用 RTAI-linux 的 patch，从而增强 uClinux 的实时性，使得 uClinux 可以应用于工业控制、进程控制等一些实时要求较高的应用。

uClinux 的内存管理

uClinux 同标准 Linux 的最大区别就在于内存管理，同时也由于 uClinux 的内存管理引发了一些标准 Linux 所不会出现的问题。本文将把 uClinux 内存管理同标准 Linux 的那内存管理部分进行比较分析。

标准 Linux 使用的虚拟存储器技术

标准 Linux 使用虚拟存储器技术，这种技术用于提供比计算机系统中实际使用的物理内存大得多的内存空间。使用者将感觉到好像程序可以使用非常大的内存空间，从而使得编程人员在写程序时不用考虑计算机中的物理内存的实际容量。

为了支持虚拟存储管理器的管理，Linux 系统采用分页（paging）的方式来载入进程。所谓分页既是把实际的存储器分割为相同大小的段，例如每个段 1024 个字节，这样 1024 个字节大小的段便称为一个页面（page）。

虚拟存储器由存储器管理机制及一个大容量的快速硬盘存储器支持。它的

实现基于局部性原理，当一个程序在运行之前，没有必要全部装入内存，而是仅将那些当前要运行的那些部分页面或段装入内存运行（copy-on-write），其余暂时留在硬盘上程序运行时如果它所要访问的页（段）已存在，则程序继续运行，如果发现不存在的页（段），操作系统将产生一个页错误（page fault），这个错误导致操作系统把需要运行的部分加载到内存中。必要时操作系统还可以把不需要的内存页（段）交换到磁盘上。利用这样的方式管理存储器，便可把一个进程所需要用到的存储器以化整为零的方式，视需求分批载入，而核心程序则凭借属于每个页面的页码来完成寻址各个存储器区段的工作。

标准 Linux 是针对有内存管理单元的处理器设计的。在这种处理器上，虚拟地址被送到内存管理单元（MMU），把虚拟地址映射为物理地址。

通过赋予每个任务不同的虚拟——物理地址转换映射，支持不同任务之间的保护。地址转换函数在每一个任务中定义，在一个任务中的虚拟地址空间映射到物理内存的一个部分，而另一个任务的虚拟地址空间映射到物理存储器中的另外区域。计算机的存储管理单元（MMU）一般有一组寄存器来标识当前运行的进程的转换表。在当前进程将 CPU 放弃给另一个进程时（一次上下文切换），内核通过指向新进程地址转换表的指针加载这些寄存器。MMU 寄存器是有特权的，只能在内核态才能访问。这就保证了一个进程只能访问自己用户空间内的地址，而不会访问和修改其它进程的空间。当可执行文件被加载时，加载器根据缺省的 ld 文件，把程序加载到虚拟内存的一个空间，因为这个原因实际上很多程序的虚拟地址空间是相同的，但是由于转换函数不同，所以实际所处的内存区域也不同。而对于多进程管理当处理器进行进程切换并执行一个新任务时，一个重要部分就是为新任务切换任务转换表。我们可以看到 Linux 系统的内存管理至少实现了以下功能：

运行比内存还要大的程序。理想情况下应该可以运行任意大小的程序

可以运行只加载了部分的程序，缩短了程序启动的时间

可以使多个程序同时驻留在内存中提高 CPU 的利用率

可以运行重定位程序。即程序可以方于内存中的任何一处，而且可以在执行过程中移动。
写机器无关的代码。程序不必事先约定机器的配置情况。
减轻程序员分配和管理内存资源的负担。
可以进行共享——例如，如果两个进程运行同一个程序，它们应该可以共享程序代码的同一个副本。
提供内存保护，进程不能以非授权方式访问或修改页面，内核保护单个进程的数据和代码以防止其它进程修改它们。否则，用户程序可能会偶然（或恶意）的破坏内核或其它用户程序。
虚存系统并不是没有代价的。内存管理需要地址转换表和其他一些数据结构，留给程序的内存减少了。地址转换增加了每一条指令的执行时间，而对于有额外内存操作的指令会更严重。当进程访问不在内存的页面时，系统发生失效。系统处理该失效，并将页面加载到内存中，这需要极耗时间的磁盘 I/O 操作。总之内存管理活动占用了相当一部分 cpu 时间（在较忙的系统中大约占 10%）。
uClinux 针对 NOMMU 的特殊处理
对于 uClinux 来说，其设计针对没有 MMU 的处理器，即 uClinux 不能使用处理器的虚拟内存管理技术（应该说这种不带有 MMU 的处理器在嵌入式设备中相当普偏）。uClinux 仍然采用存储器的分页管理，系统在启动时把实际存储器进行分页。在加载应用程序时程序分页加载。但是由于没有 MMU 管理，所以实际上 uClinux 采用实存储器管理策略（real memeory management）。这一点影响了系统工作的很多方面。
uClinux 系统对于内存的访问是直接的，（它对地址的访问不需要经过 MMU，而是直接送到地址线上输出），所有程序中访问的地址都是实际的物理地址。操作系统对内存空间没有保护（这实际上是很多嵌入式系统的特点），各个进程实际上共享一个运行空间（没有独立的地址转换表）。
一个进程在执行前，系统必须为进程分配足够的连续地址空间，然后全部载入主存储器的连续空间中。与之相对应的是标准 Linux 系统在分配内存

时没有必要保证实际物理存储空间是连续的，而只要保证虚存地址空间连续就可以了。另外一个方面程序加载地址与预期（ld 文件中指出的）通常都不相同，这样 relocation 过程就是必须的。此外磁盘交换空间也是无法使用的，系统执行时如果缺少内存将无法通过磁盘交换来得到改善。

uClinux 对内存的管理减少同时就给开发人员提出了更高的要求。如果从易用性这一点来说，uClinux 的内存管理是一种倒退，退回了到了 UNIX 早期或是 Dos 系统时代。开发人员不得不参与系统的内存管理。从编译内核开始，开发人员必须告诉系统这块开发板到底拥有多少的内存（假如你欺骗了系统，那将在后面运行程序时受到惩罚），从而系统将在启动的初始化阶段对内存进行分页，并且标记已使用的和未使用的内存。系统将在运行应用时使用这些分页内存。

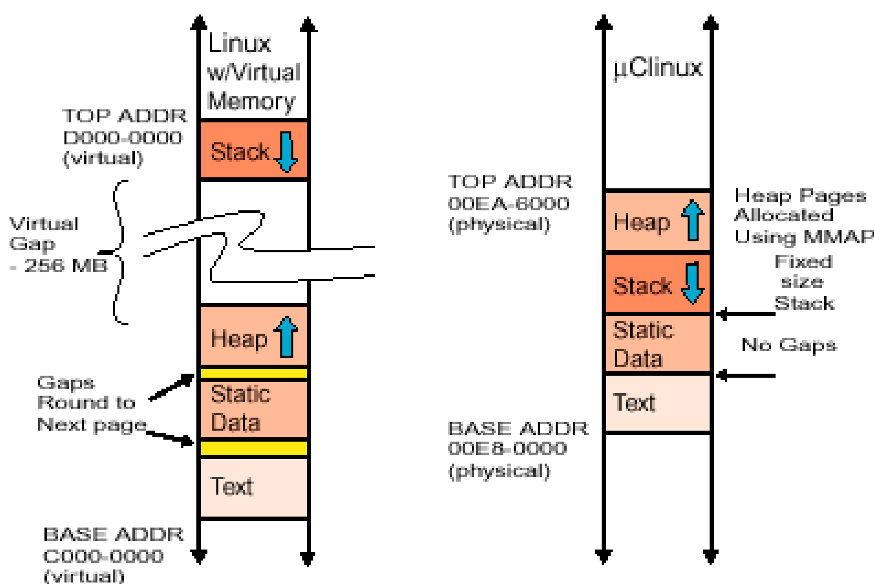
从内存的访问角度来看，开发人员的权利增大了（开发人员在编程时可以访问任意的地址空间），但与此同时系统的安全性也大为下降。此外，系统对多进程的管理将有很大的变化，这一点将在 uClinux 的多进程管理中说明。

虽然 uClinux 的内存管理与标准 Linx 系统相比功能相差很多，但应该说这是嵌入式设备的选择。在嵌入式设备中，由于成本等敏感因素的影响，普偏的采用不带有 MMU 的处理器，这决定了系统没有足够的硬件支持实现虚拟存储管理技术。从嵌入式设备实现的功能来看，嵌入式设备通常在某一特定的环境下运行，只要实现特定的功能，其功能相对简单，内存管理的要求完全可以由开发人员考虑。

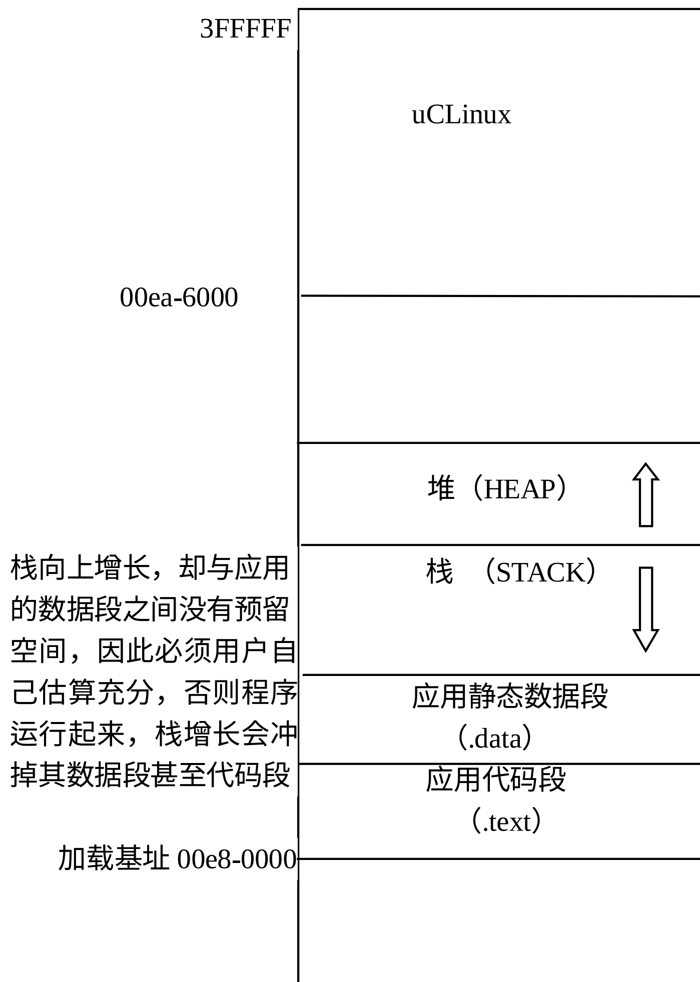
内存分布图：

下图为标准 Linux 系统和 uClinux 系统的进程内存布局图,在标准 Linux 系统中（如左图），系统数据段，代码段，堆和栈在虚存层面是连续的。堆向上增长，栈向下增长，在堆底和栈顶之间有 256MB 的内存可供分配。uClinux 采用了实内存模式，各个内存段在物理内存（没有虚存）层面都是连续的，栈段在同数据段在一起，堆有系统内存管理，所有进程共享，

由于内存连续和保护的要求（详见正文），栈段，数据段，代码段都是在程序加载是分配，栈段的大小固定（在生成应用时可以指定栈段大小），开发人员在开发时不得不使用一些方法估计判断栈段的大小，使其即能满足程序的需要，又不浪费内存。



应用在华恒开发板上跑起来后，CPU 中的内存映象：



这种内存空间布局还阻碍了动态连接库的运用，一个很简单的方面，应用程序将不知道到哪儿去寻找已被加载的动态库的位置（使用虚拟内存的方式可以利用虚拟内存的重映射表）。这个问题的一个解决办法事先给定动态库被加载物理内存的某个位置，并且确定要使用的内存空间，在系统启

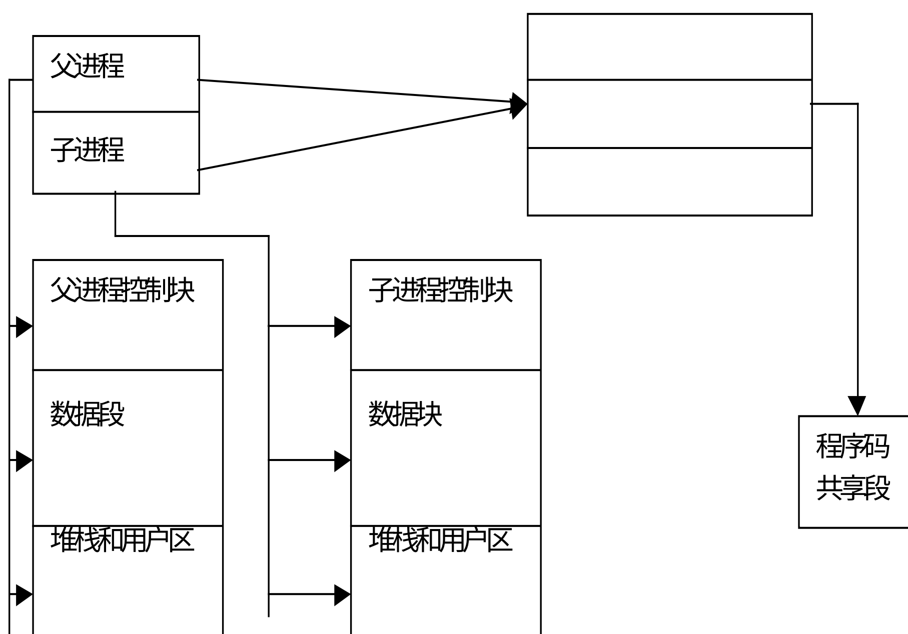
动是首先加载动态库（加载到事先假定的位置）。另一个问题是这个动态库必须是绝对可重入的，由于没有了内存保护的机制（通常由 MMU 单元提供），所有动态库内的数据需要使用局部变量，以使得这些数据分配于各个进程的栈段，如果考虑使用锁的机制实现这种共享访问可能更加复杂。

标准 Linux 系统的进程、线程

进程：进程是一个运行程序并为其提供执行环境的实体，它包括一个地址空间和至少一个控制点，进程在这个地址空间上执行单一指令序列。进程地址空间包括可以访问或引用的内存单元的集合，进程控制点通过一个一般称为程序计数器（program counter,PC）的硬件寄存器控制和跟踪进程指令序列。

fork：由于进程为执行程序的环境，因此在执行程序前必须先建立这个能"跑"程序的环境。Linux 系统提供系统调用拷贝现行进程的内容，以产生新的进程，调用 fork 的进程称为父进程；而所产生的新进程则称为子进程。子进程会承袭父进程的一切特性，但是它有自己的数据段，也就是说，尽管子进程改变了所属的变量，却不会影响到父进程的变量值。

Linux 系统中亲子进程之间的关系可用下图表示：



父进程和子进程共享一个程序段，但是各自拥有自己的堆栈、数据段、用户空间以及进程控制块。换言之，两个进程执行的程序代码是一样的，但是各有各的程序计数器与自己的私人数据。

当内核收到 fork 请求时，它会先查核三件事：首先检查存储器是不是足够；其次是进程表是否仍有空缺；最后则是看看用户是否建立了太多的子进程。如果上述说三个条件满足，那么操作系统会给子进程一个进程识别码，并且设定 cpu 时间，接着设定与父进程共享的段，同时将父进程的 inode 拷贝一份给子进程运用，最终子进程会返回数值 0 以表示它是子进程，至于父进程，它可能等待子进程的执行结束，或与子进程各做个的。

exec 系统调用：该系统调用提供一个进程去执行另一个进程的能力，exec 系统调用是采用覆盖旧有进程存储器内容的方式，所以原来程序的堆栈、

数据段与程序段都会被修改，只有用户区维持不变。

vfork 系统调用：由于在使用 fork 时，内核会将父进程拷贝一份给子进程，但是这样的做法相当浪费时间，因为大多数的情形都是程序在调用 fork 后就立即调用 exec，这样刚拷贝来的进程区域又立即被新的数据覆盖掉。因此 Linux 系统提供一个系统调用 vfork，vfork 假定系统在调用完成 vfork 后会马上执行 exec，因此 vfork 不拷贝父进程的页面，只是初始化私有的数据结构与准备足够的分页表。这样实际在 vfork 调用完成后父子进程事实上共享同一块存储器（在子进程调用 exec 或是 exit 之前），因此子进程可以更改父进程的数据及堆栈信息，因此 vfork 系统调用完成后，父进程进入睡眠，直到子进程执行 exec。当子进程执行 exec 时，由于 exec 要使用被执行程序的数据，代码覆盖子进程的存储区域，这样将产生写保护错误（do_wp_page）（这个时候子进程写的实际上是父进程的存储区域），

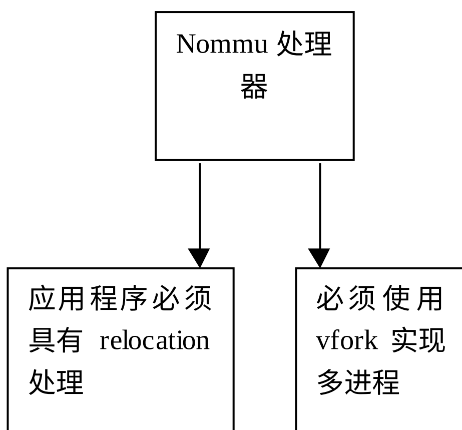
这个错误导致内核为子进程重新分配存储空间。当子进程正确开始执行后，将唤醒父进程，使得父进程继续往后执行。

uClinux 的多进程处理

uClinux 没有 mmu 管理存储器，在实现多个进程时（fork 调用生成子进程）需要实现数据保护。

uClinux 的 fork 和 vfork：uClinux 的 fork 等于 vfork。实际上 uClinux 的多进程管理通过 vfork 来实现。这意味着 uClinux 系统 fork 调用完程后，要么子进程代替父进程执行（此时父进程已经 sleep）直到子进程调用 exit 退出，要么调用 exec 执行一个新的进程，这个时候将产生可执行文件的加载，即使这个进程只是父进程的拷贝，这个过程也不能避免。当子进程执行 exit 或 exec 后，子进程使用 wakeup 把父进程唤醒，父进程继续往下执行。

uClinux 的这种多进程实现机制同它的内存管理紧密相关。uClinux 针对 nommu 处理器开发，所以被迫使用一种 flat 方式的内存管理模式，启动新的应用程序时系统必须为应用程序分配存储空间，并立即把应用程序加载到内存。缺少了 MMU 的内存重映射机制，uClinux 必须在可执行文件加载阶段对可执行文件 reloc 处理，使得程序执行时能够直接使用物理内存。



工具及内核

这是一个 **Platform port** 的过程。主要是针对自己设计的开发板，建立开发环境，运行操作系统，移植应用函数库，设计某些特定设备的驱动程序等等。

建立开发环境

如前文所述，**uClinux** 的开发环境全面的采用了 **GNU** 开发套件的解决方案。针对开发板实际上需要做的工作是在开发宿主机上建立交叉编译环境。对 **GNU** 一系列工具进行配置编译形成工具集如下：

genromfs	**arm-elf-g++**	**arm-elf-objdump**
arm-elf-addr2line	**arm-elf-gasp**	**arm-elf-ranlib**
arm-elf-ar	**arm-elf-gcc**	**arm-elf-size**
arm-elf-as	**arm-elf-gdb**	**arm-elf-strings**
arm-elf-cc1	**arm-elf-ld**	**arm-elf-strip**
arm-elf-cc1plus	**arm-elf-nm**	**elf2flt**
arm-elf-cpp	**arm-elf-objcopy**	

内核和应用程序都是采用 **ELF** 可执行文件格式，都使用 **arm-elf-gcc** 编译

器。

进程切换优化

前文已经提及 Linux 系统的进程，并且说明了系统生成进程的过程以及复杂进程之间使用的 copy-on-write 技术。这里需要更进一步提及 Linux 系统的线程。

在一个应用程序中使用多个进程有一个明显的缺点。创建这些进程增加了一些基本开销，因为 fork 是一种很大的系统调用，因为每个进程都有它自己的地址空间，它必须使用进程间通信的手段如消息传递或者共享内存。要把这些进程分配到不同的机器或处理器上去运行，及在进程之间传递消息，等待进程完成，收集结果等都需要额外的开销。由于这些原因，线程机制出现了。这时允许一个应用含有多个线程，所有这些线程共享一个运行空间，但是每个线程有各自的私有栈。当系统在管理这些多线程时（不论是用户态的线程，还是内核支持的线程），线程之间的切换将要比进程之间的切换减少很多的工作量，从而节约系统的开销。

所谓的多进程实际上共享一个运行空间，每一个进程拥有自己独立的用户栈。这种情形非常类似多线程的工作方式。

但是在 uClinux 处理多进程的时候仍然作了大量的工作，而这些工作很多是那种真正支持多进程处理时所需要做的（这些工作包括切换内存转换表等）。于是我们可以考虑优化 uClinux 的多进程处理机制，把多个进程切换的方法简化为多个线程的切换方法，使得其更加简洁高效。

附录 F 图形界面（GUI）接口函数 API

华恒 ucLinux 开发平台提供的 GUI API 仿照 WIN32 API 的接口，使客户能够以最短的时间熟悉并使用它们，实现从 WINDOWS 平台到 LINUX 平台开发者的角色转变。

short initgraph(void)

初始化显示环境,返回结果表示初始化是否成功

void closegraph(void)

关闭显示环境

void clearscreen(void)

清屏

void setpixel(short x,short y,short color)

在（x,y）坐标处画点

short getpixel(short x,short y)

返回（x,y）点的颜色信息

void setmode(CopyMode mode)

设置显示模式

参数 mode 可为下值：

MODE_SRC	从源到目的 COPY
MODE_NOT_SRC	目的为源的反
MODE_SRC_OR_DST	目的为与源相或后的结果
MODE_SRC_AND_DST	目的为与源相与后的结果
MODE_SRC_XOR_DST	目的为与源相异或后的结果
MODE_SRC_OR_NOT_DST	目的为与源的反相或后的结果

MODE_SRC_AND_NOT_DST　目的为目的的反与源相与后的结果
MODE_SRC_XOR_NOT_DST　目的为目的的反与源相异或后的结果
MODE_NOT_SRC_OR_DST　目的为与源的反相或后的结果
MODE_NOT_SRC_AND_DST　目的为与源相与后的结果
MODE_NOT_SRC_XOR_DST　目的为与源的反相异或后的结果
MODE_NOT_SRC_OR_NOT_DST 目的取反与源的反相或后的结果
MODE_NOT_SRC_AND_NOT_DST 目的取反与源的反相与后的结果
MODE_NOT_SRC_XOR_NOT_DST 目的取反与源的反相异或后的结果

CopyMode getmode(void);
　获取当前显示模式

void setcolor(short color)
　设置显示前景色

UINT getcolor(void)
　获取当前显示前景色

void setfillpattern(PatternIndex index)
　设置填充模式

PatternIndex getfillpattern(void)
　获取当前填充模式

void bar(short x1,short y1,short x2,short y2)
　以实填充模式在（x1,y1,x2,y2）绘制矩形

void ellipse(short x1,short y1,short x2,short y2)
在矩形（x1,y1,x2,y2）中绘制椭圆

void line(short x1, short y1, short x2,short y2)
　从点（x1,y1）到点（x2,y2）画一条直线

void lineto(short x, short y)　从当前所在点到点（x,y）之间画一条直线

void moveto(short x,short y)　设置当前所在点为点（x,y）

void rectangle(short x1,short y1,short x2,short y2)
　在（x1,y1,x2,y2）中绘制矩形

void textout(short x,short y,unsigned char *s)

在（x,y）坐标处输出字符串 s

void bitblt(
    short src_x,
    short src_y,
    short w,
    short h,
    short dest_x,
    short dest_y,
    unsigned char *src,
    short src_units_per_line,
    unsigned char *dest,
    short dest_units_per_line
)

位块传送，源地址(src_x,src_y),宽度 w,高度 h，目标地址(dest_x,dest_y)，源块的数据指针 src，src_units_per_line 指明了数据源中每行的宽度，dest 指明了目的数据地址，dest_units_per_line 指明了目的地址方的每行的宽度

void ShowBMP(char *filename,short x,short y)

在 x，y 处显示位图

void draw_bmp(short sx, short sy, short rwidth, short height, char* pixel)

位图显示函数在指定的 sx,sy 处显示每行逻辑宽度为 rwidth 字节的位图，pixel 中指定了位图的图象信息，通常的使用可以参考 ShowBMP 中的使用

void V_scroll_screen(short height)

竖直方向滚动屏幕，以 height 为单位。height>0，屏幕上滚；height<0，屏幕下滚。

void H_scroll_screen(short height)

水平方向滚动屏幕，以 height 为单位。height>0，屏幕向左滚动；

height<0，屏幕向右滚动。

附录 G 参考资料

A) W90N740B RISC Microcontroller－User’s Manual，Rev1
http://www.samsung.com/Products/Semiconductor/SystemLSI/Networks/PersonalNTASSP/CommunicationProcessor/W90N740B/W90N740B.htm

B) 《UNIX 环境高级编程》，W.Richard Stevens 著，机械工业出版社

C) 《LINUX 设备驱动程序》，ALESSANDRO RUBINI 著，LISOLEG 译，中国电力出版社

D) 《嵌入式 LINUX 设计与应用》（清华大学出版社 2002 年出版）